



UNIVERSIDAD DE MÁLAGA



GRADUADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN TECNOLOGÍAS DE LA INFORMACIÓN

DISEÑO DE UNA ARQUITECTURA BASADA EN CONTENEDORES PARA LA INTEGRACIÓN Y EL DESPLIEGUE CONTINUO (CI/CD)

DESIGN OF A CONTAINER-BASED ARCHITECTURE FOR THE INTEGRATION AND CONTINUOUS DELIVERY (CI/CD)

Realizado por

Denis Maggi

Tutorizado por

Eladio Gutiérrez Carrasco

Departamento

Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA

MÁLAGA, FEBRERO DE 2020

Resumen

En este trabajo se ha diseñado e implementado una arquitectura para el desarrollo de aplicaciones con el uso de la tecnología de contenedores, dotándola de Integración Continua (CI) mediante la automatización de tareas con *shell scripts* y el uso de servicios externos, así como de Despliegue Continuo (CD) con la herramienta *Jenkins*.

En primer lugar se ha elegido una aplicación ejemplo, analizando qué recursos y configuraciones requiere (e.g., ficheros) para su integración en contenedores. Esta aplicación sería la que está en desarrollo en un contexto de una empresa de software.

Tras la elección, se ha estructurado el proyecto según los diferentes servicios que componen la aplicación.

Se han realizado las configuraciones y el desarrollo para servir la aplicación en una máquina local con el fin de seguir cambiando y depurando la aplicación. Además de asegurar un despliegue seguro, escalable y de alta disponibilidad para el público en servidores remotos.

También se ha especificado y recomendado un flujo de trabajo con el que poder aplicar las diferentes técnicas empleadas, tales como, el versionado de archivos, versionado de imágenes, automatización del despliegue, etc.

Por último se integran servicios adicionales para la gestión y el mantenimiento del servicio.

Palabras clave: contenedores, arquitectura basada en la nube, escalado horizontal, integración continua, despliegue continuo, integración.

Abstract

In this work we have designed and implemented an architecture for cloud-based applications with the use of container technology, providing it with Continuous Integration (CI) by automating tasks with shell scripts and using external services, as well as Continuous Delivery (CD) with the *Jenkins* tool.

First, an example application has been chosen, analyzing the resources and configurations that it requires with the goal to be integrated into containers. This application would be this one that is under development in the context of a software company.

After selecting the sample application, the project has been structured according to the different services included in the application.

To deploy the application on a local system, all the settings and installations have been performed. The goal is carrying out a continuous development, changing and debugging of the application. Additionally, a secure, scalable, and highly available deployment to the public on remote servers is assured.

A workflow has also been specified and recommended where it is applied all the different techniques used, such as file versioning, image versioning, deployment automation, etc.

Finally, additional services are integrated for the management and maintenance of the service.

Keywords: containers, cloud architecture, horizontal scaling, continuous integration, continuous delivery, integration.

Índice general

Resumen	I
Abstract	III
1. Introducción	1
1.1. Estado del Arte	1
1.1.1. Virtualización y Contenedores	1
1.1.1.1. Virtualización vs Contenedores	3
1.1.2. Integración Continua y Despliegue Continuo	3
1.1.3. Escalabilidad y Alta Disponibilidad	4
1.2. Contexto y Justificación del trabajo	6
1.3. Objetivo	7
2. Entorno Tecnológico	9
2.1. Lenguajes de programación	9
2.2. Herramientas	10
2.3. Servicios	15
3. Desarrollo e Implementación	17
3.1. Descripción de la aplicación ejemplo	17
3.1.1. Desarrollo tradicional frente a las nuevas tendencias	19
3.1.2. Agentes interventores	20
3.2. Descripción del entorno	21
4. Arquitectura Local	23
4.1. Imágenes Docker	23
4.2. Despliegue de contenedores Docker	25
4.3. Scripts	28
4.3.1. Configuraciones	36
4.4. Infraestructura de la red	36
4.5. Flujo de trabajo en Local	37
4.6. Mantenimiento y nuevo desarrollo	37
5. Arquitectura Cloud	39
5.1. Nueva estructura de la aplicación	39
5.2. Imágenes Docker	40
5.3. Descripción de servicios	41

5.3.1.	Nuestra aplicación como servicio - tfgdenis	41
5.3.2.	Servicio - monitoring	42
5.3.3.	Servicio - visualizer	43
5.3.4.	Servicio - jenkins	46
5.4.	Despliegue de contenedores Docker	51
5.4.1.	Servicio - tfgdenis	51
5.4.2.	Servicio - monitoring	52
5.4.3.	Servicio - jenkins	54
5.4.4.	Servicio - visualizer	56
5.5.	Orquestación - Docker Swarm	56
5.6.	Scripts y Configuraciones	57
5.6.1.	Configuración de dominios	57
5.6.2.	Configuración ingress - tfgdenis	59
5.6.3.	Configuración Prometheus - monitoring	61
5.7.	Infraestructura de la red	62
5.7.1.	Servicio tfgdenis	63
5.7.2.	Servicio monitoring	63
5.7.3.	Servicio jenkins	63
5.8.	Seguridad	65
5.9.	Flujo de trabajo en el servidor	66
5.9.1.	Alta disponibilidad y la escalabilidad	66
5.9.2.	Despliegue Continuo (CI/CD)	67
5.10.	Mantenimiento y nuevo desarrollo	67
6.	Conclusiones	69
	Anexos	70
A.	Instalación de Herramientas	73
A.1.	<i>docker</i> y <i>docker-compose</i>	73
A.1.1.	Instalación Entorno Local	73
A.1.1.1.	docker	73
A.1.1.2.	docker-compose	74
A.1.2.	Instalación Entorno Servidores/Cloud	74
A.1.3.	Uso de Docker y docker-compose	74
A.2.	Instalación de TSRC	75
A.3.	Uso y configuración de TSRC	76
B.	Configuración de Herramientas	77
B.1.	Configuración <i>.bashrc</i>	77
B.1.0.1.	Entorno Local	77
B.1.0.2.	Entorno en Servidores	77
B.2.	Configuración SSH	78
B.2.0.1.	Entorno Local	78
B.2.0.2.	Entorno en Servidores	79
B.3.	Configuración Jenkins	80
B.3.1.	Configuración inicial	80

B.3.2.	Configuración Gitlab	81
B.3.2.1.	Configuración SSH	81
B.3.2.2.	Configuración GitLab API	82
B.3.3.	Configuración Docker Hub	82
C.	Configuración de Servicios	83
C.1.	Configuración GitLab	83
C.1.1.	Configuración SSH	83
C.1.2.	Configurar WebHooks	84
C.2.	Configuración DockerHub	84
C.3.	Configuración DigitalOcean	85
C.4.	Configuración DinaHosting	85
Referencias		89

Capítulo 1

Introducción

1.1. Estado del Arte

En este capítulo explicaremos que es la virtualización, cuál es su uso más frecuente e indagaremos más en profundidad acerca de los contenedores, comparando ambas técnicas, también veremos cómo usando estas técnicas podemos aplicar los conceptos de escalabilidad y alta disponibilidad; además, veremos en que consiste la Integración Continua y el Despliegue Continuo. Por último pondremos en contexto el trabajo, así como una justificación del mismo y los objetivos finales esperados.

1.1.1. Virtualización y Contenedores

Virtualización Es el procedimiento de crear *software* para **emular recursos tecnológicos** tales como un servidor, un sistema operativo, un dispositivo de almacenamiento o cualquier otro recurso de red. En otras palabras lo que se hace es una abstracción de los recursos del *hardware* llamada VMM (*Virtual Machine Monitor* o Hipervisor) que crea una capa intermedia entre la máquina física *host* y el sistema operativo de la máquina virtual *guest*, repartiendo el recurso en uno o más entornos de ejecución. Como resultado tendremos varias máquinas virtuales *ejecutándose* en el mismo computador físico. Pero esta configuración tiene como consecuencia la necesidad de crear servidores cada vez con capacidad para poder soportar múltiples sistemas virtualizados simultáneamente.

El concepto de **hipervisor** nace en los años 60 como una generalización del *supervisor* que se aplicaba a los *kernels*¹ de los Sistemas Operativos y como respuesta al uso intensivo de los poderosos *mainframes*² creados por fabricantes como *IBM* con la intención de reducir los costos que generaban muchos ordenadores trabajando aislados en la misma empresa.

¹Núcleo de los Sistemas Operativos

²Supercomputadora que ofrece servicios de red a varios clientes

No obstante su popularización y uso masivo ocurre en los años 90, en los que empresas como *Vmware Inc.*³ se proponen desarrollar un software de virtualización para computadores basados en procesadores Intel x86, ampliamente extendidos gracias a la arquitectura PC que no solo estaba presente en los ordenadores domésticos sino también, y cada vez más, en la gama de servidores. Es de destacar que la arquitectura Intel IA-32 no tenía de partida soporte hardware para virtualización y, dada sus particularidades, un sistema de virtualización completa representaba todo un reto. Es en este momento cuando comienza a popularizarse la virtualización como ahora la conocemos. Sin embargo, estos hipervisores de máquinas completas, que emulan el *hardware* físico para que las máquinas virtuales puedan utilizarlo, son ávidos de recursos. Ello causa ciertas limitaciones como, por ejemplo, que las imágenes de las máquinas virtuales resulten muy *pesadas* de arrancar (*Virtualización*, 2020).

Contenedores La pregunta ¿por qué instalar un sistema operativo en cada máquina virtual? Nos lleva a la respuesta obvia: porque cada máquina virtual es una entidad aislada e independiente de las demás. La consecuencia es que necesitan más recursos (espacio en disco, disponibilidad de procesamiento y memoria). Para dar otra respuesta a esta configuración que devora más y más recursos, nos preguntamos **¿será posible ejecutar gran cantidad de aplicaciones (encapsuladas, empaquetadas) en el mismo Sistema Operativo?**. La respuesta es sí. De esta necesidad nace el concepto de **contenedor** en el contexto de la virtualización (Rodríguez, s.f.; Security, s.f.).

Un primer precedente histórico lo encontramos en el mecanismo *chroot*, introducido durante el desarrollo de *Unix V7* en 1970. A modo de *sandbox*, este mecanismo permite cambiar el directorio raíz a un proceso y sus hijos, de modo que el nuevo directorio raíz actúa de *jaula* donde dichos procesos quedan *encerrados*. Los contenedores modernos básicamente amplían este concepto. En 2008 nace LXC (*Linux Containers*) la primera y más completa implementación para gestionar contenedores de *Linux*. Y en 2013 emerge *Docker and The Future* y explota su popularidad. El crecimiento de Docker y el uso de contenedores, van de la mano (*Una Breve Historia de los Contenedores: desde los años 70 a Docker 2016*, 2017).

Cuando pensamos en contenedores se nos viene a la mente el inmenso océano surcado de grandes barcos con pilas y pilas de esas cajas metálicas llenas de mercancía que se transporta a todas partes del globo. De manera similar los Contenedores como herramienta de virtualización utilizan *software* a modo de barco-plataforma (LXC, Docker,...) aprovechando los recursos del computador (procesador, memoria, red) sobre un mismo Sistema Operativo.

Así pues, los contenedores comparten el mismo *kernel* del Sistema Operativo y están enfocados a poder ejecutar aplicaciones y sus servicios relacionados de forma independiente y siempre buscando que se comporten de igual forma en todos los entornos. Las partes compartidas del Sistema Operativo son de solo lectura, mien-

³Esta compañía fue pionera en la virtualización de la arquitectura Intel IA-32 con su patente de traducción binaria de 1998 registrada como *U.S. Patent 6,397,242*

tras que cada contenedor tiene su propio montaje para la escritura. Esto significa que los contenedores son mucho más *livianos* y utilizan, en conjunto, menos recursos (Rodríguez, s.f.; Security, s.f.).

1.1.1.1. Virtualización vs Contenedores

En la Tabla 1.1 se comparan las ventajas de utilizar una tecnología frente a la otra.

	VIRTUALIZACIÓN	CONTENEDORES
Principales Ventajas	<ul style="list-style-type: none"> ■ Permite entornos de prueba, fuerte aislamiento y seguridad. ■ Mejora las políticas de copias de seguridad. ■ Clonación y migración de sistemas en caliente. 	<ul style="list-style-type: none"> ■ Independencia de la plataforma. ■ Instalación mas sencilla. Se instalan mediante una imagen que no requiere de configuración extra. ■ Pérdidas por virtualización mínimas y aplicaciones aisladas. ■ Administración y automatización unitarias.

Tabla 1.1: Máquinas virtuales vs. contenedores

Concluimos entonces que ambas tecnologías se van a utilizar para fines distintos, es decir, **los contenedores no vienen a sustituir a las máquinas virtuales**. De hecho, como se muestra en la Figura 1.1 **ambas tecnologías son totalmente compatibles** y lo más usual es **utilizarlas juntas para generar una infraestructura mas potente, segura, flexible y eficiente**.

1.1.2. Integración Continua y Despliegue Continuo

Continuous Integration - CI Los desarrolladores que aplican la CI fusionan sus cambios con la mayor frecuencia posible. Los cambios del desarrollador se validan compilando y ejecutando pruebas automatizadas. Haciendo esto se evita la dura etapa de integración, la cual ocurre generalmente el día del lanzamiento, que es cuando las personas vuelcan sus cambios. La CI pone gran énfasis en la automatización de

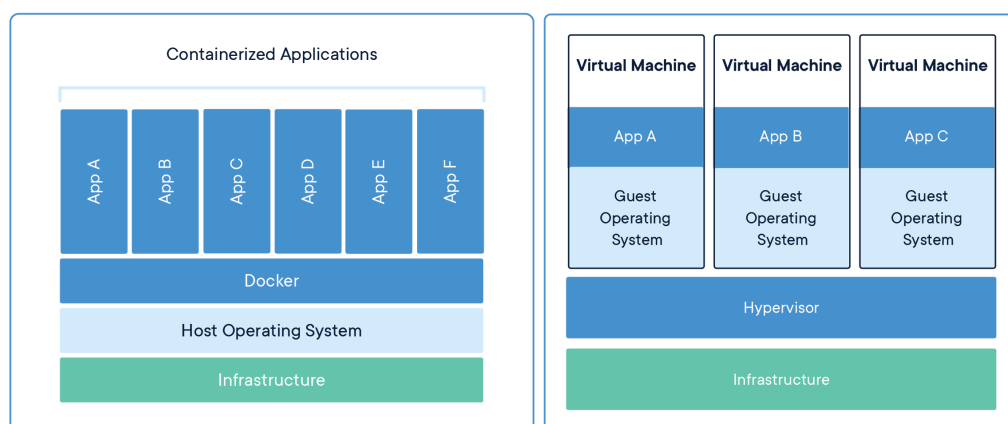


Figura 1.1: Arquitectura interna de contenedores vs máquinas virtuales.

Fuente: (*What is a Container?*, s.f.)

pruebas para asegurar de que la aplicación *no se rompe* cada vez que se integran nuevos cambios (Atlassian, s.f.).

Por tanto, lo que necesitaremos para aplicar CI en nuestro proyecto será:

- Escribir pruebas automatizadas para cada mejora o corrección de errores.
- Un servidor de integración continua que monitoree el repositorio principal y ejecute las pruebas automáticamente para cada nueva confirmación enviada.
- Los desarrolladores deben consolidar sus cambios periódicamente, por ejemplo, al menos una vez al día.

Continuous Delivery - CD El CD se puede considerar como una extensión de la Integración Continua, y su objetivo es asegurarnos de que podremos generar los nuevos cambios del sistema y entregarlos rápidamente al usuario final de manera sustancial. Esto significa que antes de automatizar las pruebas, ya están automatizados los procesos de emisión de nuevas versiones y por tanto se puede ejecutar la aplicación. En teoría con CD se puede decidir cuándo emitir nuevas versiones, si diariamente, semanalmente, ... dependiendo de los requerimientos de la empresa. Lo aconsejable para obtener beneficios reales de la CD, es implementarlo en producción lo antes posible para asegurarse de trabajar con pequeños lotes que sean fáciles de depurar en caso de problemas. (Atlassian, s.f.)

Por tanto, para aplicar CD en nuestro proyecto, necesitaremos:

- Una base sólida en la CI y un conjunto de pruebas que cubra suficiente código.
- Automatizar las implementaciones.
- Adoptar, por parte del equipo, indicadores de características para que las implementaciones incompletas no afecten a los clientes en producción.

1.1.3. Escalabilidad y Alta Disponibilidad

La **escalabilidad** define la **capacidad de un sistema para crecer en magnitud**.

Es una propiedad que se espera tengan los sistemas para que puedan reaccionar al crecimiento de trabajo continuo adaptándose sin perder calidad. La escalabilidad como propiedad de un sistema es difícil de definir, más bien deberíamos definir los requisitos que se necesitan para que sea escalable.

Existen dos tipos que definimos a continuación cuyas principales ventajas y desventajas se resumen en la Tabla 1.2 (*Escalabilidad*, 2020).

	VENTAJAS	DESVENTAJAS
Escalado Vertical	<ul style="list-style-type: none"> ▪ Facilidad de implementación y configuración. ▪ No requiere un diseño específico en la aplicación ▪ Su arquitectura para funcionar. 	<ul style="list-style-type: none"> ▪ Está limitado por el hardware. ▪ No aporta alta disponibilidad. ▪ Hacer un <i>upgrade</i> del hardware puede ser muy costoso
Escalado Horizontal	<ul style="list-style-type: none"> ▪ El escalado es prácticamente <i>infinito</i>. ▪ Permite alta disponibilidad. ▪ Se puede combinar con el escalado vertical. ▪ Permite un correcto balanceo de carga entre los servidores. ▪ Si un nodo falla, los demás siguen trabajando. 	<ul style="list-style-type: none"> ▪ Requiere mucho mantenimiento. ▪ Necesita que la aplicación esté construida de modo que soporte escalamiento horizontal. ▪ Suele ser una opción menos económica, requiere de un desarrollo más grande.

Tabla 1.2: Comparativa escalabilidad vertical vs. horizontal

Escalado Vertical Facultad de aumentar el *hardware* existente en un nodo agregando recursos. Como vemos en la Figura 1.2 representa al bloque único más grande. Por ejemplo agregamos más memoria a un ordenador, o bien damos más potencia a un servidor, pero también podemos migrar el *hardware* completo a uno más potente. Todo esto sin que haya repercusiones sobre el software. Ese tipo de escalabilidad presenta algún aspecto negativo, teniendo en cuenta que el *hardware* tiene un límite, el crecimiento se detendría en algún momento e intentar solucionar esto puede salir muy caro no compensando el rendimiento buscado. Sin embargo, no podemos decir que este tipo de escalabilidad sea malo, sino que lo mejor es combinarlo con el escalado horizontal para obtener mejores resultados. (*Escalabilidad*, 2020)

Escalado Horizontal Es un tipo de escalabilidad más difícil de implementar y administrar. Se basa **en la modularidad y funcionalidad**. Está representada en la Figura 1.2 por múltiples bloques pequeños. En general, se añaden más equipos para dar más potencia a la red de trabajo, entendiendo potencia como la suma de la velocidad física de cada equipo transferida por la partición de aplicaciones y datos a través de los nodos. Con este modelo de escalabilidad se dispone de un sistema al que se pueden agregar recursos de manera *infinita* (*Escalabilidad*, 2020).

La **alta disponibilidad** HA (*High Availability*) es la capacidad de un sistema, o componente del sistema, de estar continuamente operativo. La disponibilidad se puede medir en relación a su *100 % de operabilidad o cero fallos*.

En tecnología informática hay un estándar ampliamente extendido (pero difícil de lograr), que se conoce como *disponibilidad 5 nueves* (99.999 %). Los expertos en disponibilidad dicen que, para que cualquier sistema esté altamente disponible, las partes de un sistema deben estar bien diseñadas y probadas exhaustivamente antes de ser utilizadas.

La planificación de la alta disponibilidad se centra en el procesamiento de copias de seguridad y conmutación de errores, el almacenamiento y el acceso a los datos. Un sistema de alta disponibilidad debería poder recuperarse rápidamente, independientemente del tipo de estado de fallo en el que se encuentre, con el objetivo de minimizar interrupciones para el usuario final.

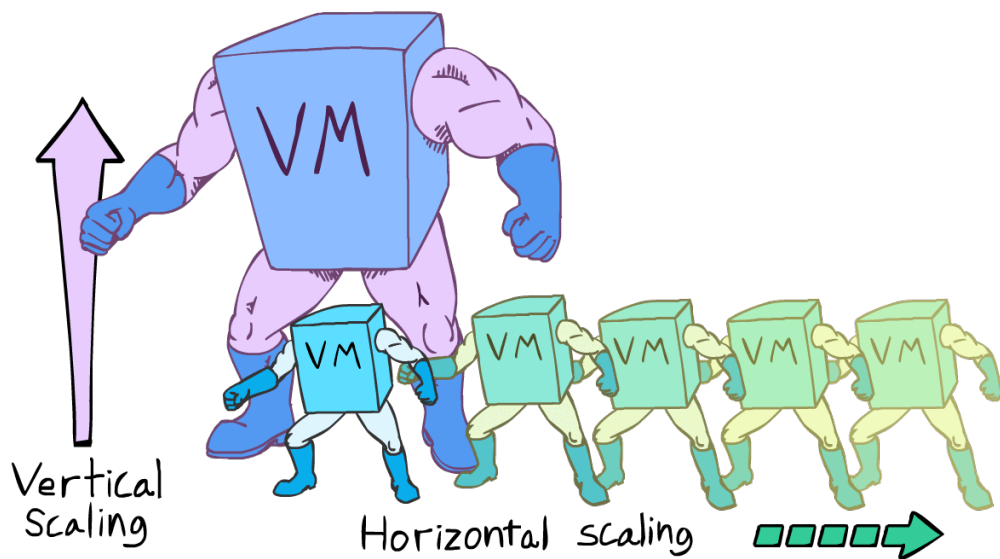


Figura 1.2: Escalabilidad Vertical frente a la Escalabilidad Horizontal.
Fuente: (*Database Scaling : Horizontal and Vertical Scaling*, s.f.)

Las copias de seguridad y los procesos de recuperación de errores son componentes cruciales para lograr una HA. Esto es porque algunos sistemas informáticos o redes consisten en componentes individuales (*hardware* o *software*) que deben estar completamente operativos para que todo el sistema esté disponible (*What is High Availability? - Definition from WhatIs.com*, s.f.).

1.2. Contexto y Justificación del trabajo

Las pequeñas y/o medianas empresas junto con las *Startups* son pequeños modelos de negocio en los que uno de los principales puntos para alcanzar el éxito y crecer es el dinero del que disponen. Por consiguiente suelen trabajar con el número mínimo de empleados para asegurar cumplir presupuestos. Además suele ser más económico contratar a personas especializadas en un campo que tener un empleado *multifunción*.

Es por ello que optar por un proceso de automatización y de desarrollo de una infraestructura que sea capaz de crecer y evolucionar conforme los requisitos de la empresa, teniendo únicamente una persona a cargo y que los demás trabajadores se vean beneficiados haciendo uso de ésta, sin tener conocimientos adicionales, supone un movimiento inteligente por parte de la empresa.

Mediante el desarrollo de la infraestructura que se plantea en este trabajo podremos dotar a la empresa de una arquitectura sólida, con facilidad para su crecimiento y su posterior mantenimiento.

1.3. Objetivo

El objetivo principal del trabajo es brindar a la pequeña o mediana empresa la facilidad de trabajar en un entorno de desarrollo lo más automatizado posible, sin tener que formar a sus trabajadores especializados en otras ramas de la informática y a su vez ofrecer una infraestructura de cara al posible crecimiento futuro de la empresa, dotándola de unas bases sólidas para el escalado vertical, horizontal y alta disponibilidad de su aplicación *Cloud*. Para ello se usará la flexibilidad y potencia que nos ofrecen los contenedores, junto a herramientas que nos ayudarán a consolidar una arquitectura segura y fiable para nuestra aplicación/servicio.

Capítulo 2

Entorno Tecnológico

2.1. Lenguajes de programación

Bash Es el *shell* de usuario predeterminado en la mayoría de las instalaciones de *Linux*. El objetivo principal de un *shell* de *Unix* es permitir que los usuarios interactúen de manera efectiva con el sistema a través de la línea de comandos. Aunque ***Bash* es principalmente un intérprete de comandos, también es un lenguaje de programación**. *Bash* admite variables, funciones y tiene construcciones de control de flujo con sentencias condicionales y bucles (*Understanding Bash: Elements of Programming / Linux Journal*, s.f.).

Todos los *shells Unix*, incluido *Bash*, son principalmente intérpretes de comandos; esta característica se remonta al primer *shell* de *Unix*, que con el tiempo han ido adquiriendo capacidades de programación por evolución (*Understanding Bash: Elements of Programming / Linux Journal*, s.f.).

Se podría decir que es primordial para el trabajo a desarrollar, pues con este lenguaje describimos los diferentes scripts para la automatización de las tareas, así como la ejecución de muchos otros programas, tales como `docker`, `tsrc`, `ssh`, etc.

Node.js Se trata de un **lenguaje de programación basado en eventos**, lo que permite el desarrollo de servidores web muy rápidos en *JavaScript*. Los desarrolladores crean servidores escalables sin utilizar subprocesos, mediante el uso de un modelo simplificado basado en eventos que utiliza retornos de llamadas para indicar que una tarea ha finalizado. En resumen, *Node.js* conecta la facilidad de un lenguaje de script (JavaScript) con el poder de la programación de red *Unix* (*Node.js*, 2019),

Python Este lenguaje de programación orientado a objetos es muy fácil de utilizar sin que por ello pierda potencia. Es de código abierto y sus adeptos lo describen como *elegante, limpio y minimalista*. Con *python* se programan *scripts* por lo que utiliza intérprete en vez de compilador. Permite programar en múltiples formas, programación multiparadigma: estructurada, funcional u

orientada a objetos o orientada a aspectos, pero también permite soporte para la programación funcional siguiendo a *Lisp* (*Python (programming language)* - *Wikipedia*, s.f.).

Java Definiendo formalmente a Java diremos que es un **lenguaje de programación de propósito general, concurrente, basado en clases, orientado a objetos** y específicamente diseñado para tener la menor cantidad posible de dependencias de implementación. Su objetivo es permitir que los desarrolladores *escriban una vez, y ejecuten en cualquier lugar* (WORA, *Write Once, Run anywhere*), lo que significa que el código *Java* compilado **puede ejecutarse en todas las plataformas que admiten java sin necesidad de volver a compilar**. WORA se logra compilando java en un lenguaje intermedio llamado *bytecode*. Una máquina virtual, llamada JVM (*Java Virtual Machine*) se encarga de ejecutar el código en cada plataforma (*What is Java programming language?* - *HowToDoInJava*, s.f.).

2.2. Herramientas

Docker Docker es una plataforma de virtualización a nivel de Sistema Operativo. Permite crear una aplicación y empaquetarla junto con sus dependencias y librerías en un contenedor, que posteriormente será capaz de ejecutarse en cualquier otra máquina que disponga de una capa para la gestión de dichos contenedores (*What is a Container?*, s.f.).

Usando Docker no es necesario el realizar pruebas en cada entorno en el que se va a ejecutar la aplicación, ni siquiera hay que hacer reconfiguraciones o ajustes. Todo se basa en el concepto de imagen de contenedor. Cada contenedor es una instancia de una imagen (puedes pensar en ellas como cajas negras ejecutables). Múltiples contenedores a partir de una misma imagen serán completamente idénticos, tan solo se diferenciarán por la capa de escritura, los datos y los archivos que manejen en cada caso. Es decir: cada contenedor es una copia exacta de una imagen (*What is a Container?*, s.f.).

A continuación veremos qué compone un contenedor y que necesitamos para crearlo

Dockerfile Es un fichero que nos sirve para detallar las reglas necesarias para definir una imagen Docker. Con ella podremos desplegar contenedores.

Imagen Docker Una imagen representa la captura de un estado particular de un futuro contenedor, dentro de la imagen se especifica todo lo que compone al contenedor en estado de ejecución, es decir, definimos el sistema operativo a usar, los paquetes que necesitamos instalar, librerías, código fuente que queramos ejecutar, así como configuraciones para los servicios que establezcamos en él (*What is a Container?*, s.f.).

Las imágenes manejan herencia. Partimos de una imagen base padre que heredamos. Existen imágenes oficiales que suelen servirnos de base para

crear un servicio. También existe una comunidad bastante amplia que crea y mantiene contenedores con servicios típicos, como *Wordpress*, *phpMyAdmin*, etc. Siempre podremos modificar estas imágenes si nos brindan el fichero *Dockerfile* o bien desplegando el contenedor, realizar cambios sobre éste y hacer lo que se conoce como un *commit*, que equivale a guardar el estado de ese contenedor como imagen.

Las imágenes internamente se construyen mediante capas (*layers*), de esta manera si descargamos en nuestro sistema una imagen base, por ejemplo, *Ubuntu*, lanzamos un contenedor, realizamos cambios sobre ella y hacemos un *commit*, obtendremos una imagen personalizada. Pues bien, esta imagen en disco solo ocupa lo que le hemos agregado, pues internamente utiliza la imagen base de *Ubuntu* en la cual nos hemos basado, permitiendo un ahorro significativo en disco cuando trabajamos con ellas.

Contenedor Es un estado de ejecución de la imagen con la cual el contenedor ha sido desplegado. A partir de una imagen se pueden desplegar uno o varios contenedores y se pueden agregar o eliminar de manera dinámica. En la Figura 2.1 podemos ver el ciclo de vida de un contenedor y por los diferentes estados por los que este puede pasar.

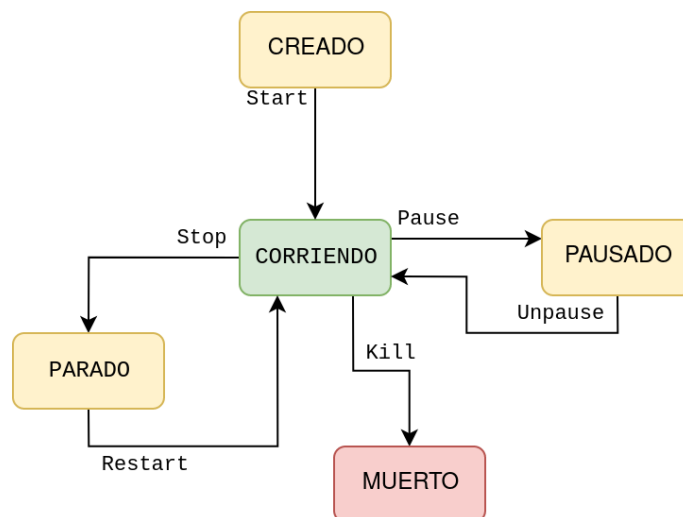


Figura 2.1: Ciclo de vida de un contenedor

Nginx Es un servidor proxy-inverso ligero y de alto rendimiento. Se trata de un *software* libre y de código abierto bajo licencia *BSD* simplificada. Es multiplataforma por lo que corre en sistemas tipo *Unix* (*GNU/Linux*, *BSD*, *Solaris*, *macOS* y *Windows*). Este sistema se utiliza en sitios web muy conocidos como: *WordPress*, *Netflix*, *GitHub*, *Source Forge* y *Facebook*. *Nginx* fue desarrollado inicialmente para satisfacer las necesidades de varios sitios web de *Rambler*, un motor de búsqueda ruso, que recibía unas 500 millones de peticiones al día en 2008 (*Nginx*, 2020).

Nginx consume muchos menos recursos que el popular servidor *Apache* y, debido a su arquitectura, puede responder a millones de peticiones por segundo, aprovechando al máximo los hilos de ejecución del servidor.

Docker Swarm Se trata de una herramienta propia de Docker que es utilizada como **orquestación de contenedores**, esto es, para administrar, monitorizar y mantener un conjunto de contenedores que se encuentran comunicados entre sí y desplegados en un *cluster* de manera centralizada.

Las actividades del *cluster* están controladas por un administrador de enjambre (*swarm manager*). Esta herramienta nos permite administrar múltiples contenedores de múltiples máquinas *host*. En un enjambre (*Docker Swarm*) generalmente hay varios nodos de trabajo, y al menos uno de administrador. El administrador es el responsable de manejar los recursos de los nodos de trabajo en forma eficiente para garantizar que el *cluster* funcione correctamente (*What is Docker Swarm?*, s.f.).

Docker Swarm reconoce tres tipos distintos de nodos, como vemos en la Figura 2.2.

- **El nodo Líder (*Leader Node*)** En el momento de crear un *cluster* se designa a uno de los nodos como líder (la asignación la realiza el algoritmo llamado RAFT¹, basado en un protocolo para implementar consenso distribuido. Su función es tomar decisiones de gestión y manejo del enjambre (*What is Docker Swarm?*, s.f.).
- **El nodo Administrador (*Manager Node*)** Cuya función principal es administrar y asignar tareas a los nodos de trabajo del enjambre. Docker recomienda un máximo de 7 nodos de administrador para un enjambre (*How nodes work*, 2020).
- **El nodo Trabajador (*Worker Node*)** Cada uno de estos nodos trabaja en función de las tareas asignadas por el administrador (por defecto, los nodos administrador, también son nodos de trabajo y pueden ejecutar tareas cuando tienen recursos disponibles para hacerlo) (*How nodes work*, 2020).

Jenkins En Ingeniería Informática las prácticas combinadas de **Integración Continua y Despliegue Continuo** fueron desarrolladas originalmente con el nombre de *Hudson* en 2004 y su primera versión se publicó en Febrero del 2005. En el 2011 se realizó una votación entre los miembros de la comunidad para cambiar el nombre de *Hudson* a *Jenkins*, y ésta fue aprobada. *Jenkins* está escrito en *Java*, bajo licencia de software libre *MIT* y se ejecuta en *GNU/Linux* (*Integración Continua de Software: Jenkins, mayordomo a nuestro servicio*, 2018).

La Integración Continua en *Jenkins* trabaja con un repositorio donde se agrupan los ficheros para que un programa o sistema pueda ser instalado en un ordenador y, a su vez, un *software* de control de versiones se encarga de su administración. Si un nuevo programador entra en el proyecto, tendremos la combinación perfecta para que comience a desarrollar de inmediato. Es importante verificar y controlar las modificaciones que se realicen, por lo que si dos programadores trabajan en el mismo código, este se volverá a compilar y

¹Para conocer cómo funciona el algoritmo visita: <http://thesecretlivesofdata.com/raft/>

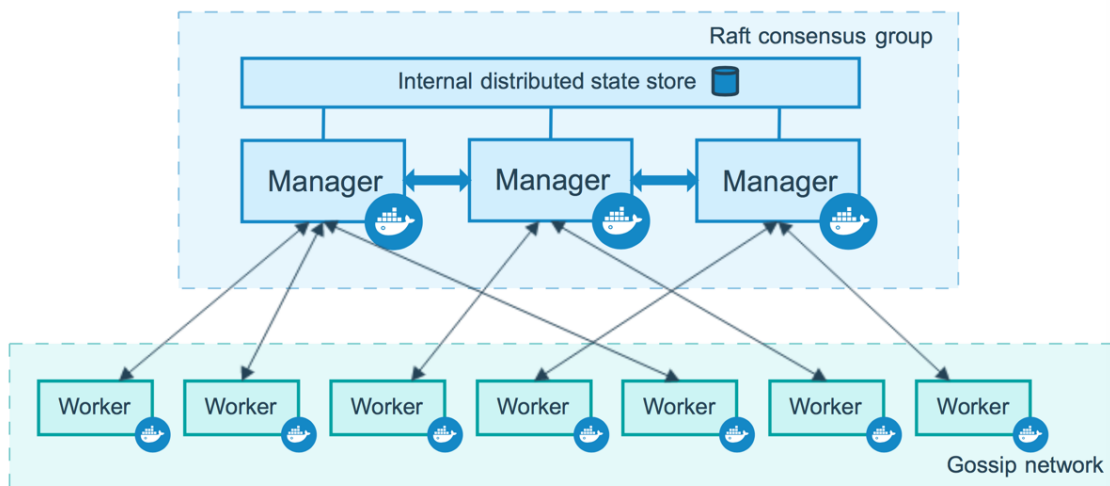


Figura 2.2: Diagrama de Docker Swarm, componentes y roles.

Fuente: (*How nodes work*, 2020)

a probar, y finalmente se subirá al repositorio con su distribución. *Jenkins* se encargará de hacer las pruebas necesarias para *aprobar* o no las modificaciones a subir (*Integración Continua de Software: Jenkins, mayordomo a nuestro servicio*, 2018).

Linux Es un **Sistema Operativo** de código abierto tipo *Unix* que se puede obtener bajo un amplio número de distribuciones.

Además de ser totalmente gratuito, *Linux* ofrece versiones para adaptarse a cualquier tipo de usuario. Pero, si se quiere construir una distribución *a medida*, *Linux* provee la ayuda *Linux From Scratch*. Además si se necesita una distribución solo para servidores que prescindan de interfaz, existen versiones de las diferentes distribuciones sin la interfaz GUI (*Graphical User Interface*) (*What is Linux?*, s.f.).

SSH y Parallel SSH (*Secure Shell*) es un protocolo de red seguro para la comunicación. Provee un canal seguro sobre un canal inseguro en una red tipo **cliente-servidor**. Es el protocolo principal que usamos para conectarnos a nuestros servidores y realizar todo tipo de tareas de manera completamente segura. *Parallel SSH* no es más que una extensión de SSH que nos permite realizar de manera paralela múltiples conexiones con múltiples destinos finales y ejecutar la misma tarea en todos ellos; esto es muy útil cuando disponemos de un *cluster* y tenemos que realizar tareas de mantenimiento y actualización en todos los nodos que lo forman.

Git Es un **sistema de control de versiones distribuido** de código abierto, lo que hace *grosso modo* es rastrear el contenido de un directorio. El código que se almacena en *Git* cambia a medida que se agrega más código, por lo que se hace imprescindible mantener un historial de cambios con características específicas (ramas y fusiones). Es un sistema de control de versiones distribuido, pues

posee un repositorio remoto almacenado en un servidor y otro local que se almacena en el ordenador de cada desarrollador, lo que se traduce a copias de código completas en cada ordenador, además del central (*An introduction to Git: what it is, and how to use it*, 2018).

El desarrollador trabaja en modo local guardando el código generado en su repositorio, pero, eventualmente, *empujará* el código, lo que se conoce como hacer *push*, a un repositorio remoto. Una vez allí todos los desarrolladores podrán ver y modificar el código. (*An introduction to Git: what it is, and how to use it*, 2018)

Prometheus Es una aplicación de *software* gratuita **utilizada para la supervisión y alerta de eventos**. Los datos de Prometheus se almacenan en forma de métricas y cada una de ellas tiene un nombre (etiqueta) que se usa para referenciarla y consultarla. Las etiquetas pueden incluir información sobre el origen de los datos (servidor del cual provienen), códigos de estado *HTTP*, métodos de consulta (*GET* y *POST*), etc. Se puede considerar que *Prometheus* es multidimensional debido a su capacidad de especificar una lista arbitraria de etiquetas y de consultas en tiempo real (admin, 2019).

Grafana Es una herramienta de código abierto para **ejecutar análisis de datos y monitorización de sistemas en línea**. Se conecta con bases de datos como *Graphite*, *Prometheus*, *Influx*, *MySQL*, etc. Podemos estudiar, analizar y monitorizar datos durante un período de tiempo en forma de *Series Temporales*, tales como el rastreo del comportamiento de usuario, como el de una aplicación, ver la frecuencia con la que aparecen errores de producción, saber su tipo y su contexto (*What is Grafana? Why Use It? Everything You Should Know About It*, 2019).

Grafana trabaja con paneles, donde se cargan los datos extraídos de las fuentes conectadas (bases de datos), ellos permiten visualizar mapas geográficos, mapas térmicos, gráficos y cuadros empresariales, etc. Generalmente, dentro de las empresas, se utiliza *Grafana* para monitorizar errores emergentes, tiempo de actividad del servidor, etc. (*What is Grafana? Why Use It? Everything You Should Know About It*, 2019)

Los paneles de *Grafana* se implementan en muchas industrias, por ejemplo *DigitalOcean* lo utiliza para compartir datos de visualización entre sus equipos, *StackOverflow* la usa para dar confiabilidad del sitio a sus desarrolladores creando paneles personalizados que optimizan el rendimiento de su servidor (*What is Grafana? Why Use It? Everything You Should Know About It*, 2019).

TSRC Se trata de una herramienta en línea de comandos para el manejo de repositorios *Git* (véase su instalación en Sección A.3). Es especialmente útil cuando manejamos muchos repositorios a la vez que y hay que realizar tareas repetitivas en cada uno de ellos, como actualizar o conocer el estado de estos (*TankerHQ/tsrc*, 2020).

LaTeX Es un sistema **de preparación de documentos**. Cuando se escribe, utiliza texto sin formato y una estructura general para el tipo de documento (artículo, libro, carta), que necesita para definir estilos (negrita, cursiva) y para agregar citas, bibliografía, notas al pie y referencias cruzadas. *LaTeX* utiliza el programa de composición tipográfica *TeX* para dar formato a su salida junto con un sistema de manejo de fuentes tipográficas (originariamente *Metafont*) (*LaTeX*, 2020).

Tanto *TeX* como *LaTeX* comenzaron como herramientas de edición para textos científicos y matemáticos, siendo utilizado desde sus inicios por académicos que necesitaban incluir en sus documentos expresiones matemáticas complejas o escrituras no latinas. *LaTeX* proporciona **un lenguaje de marcas descriptivo de alto nivel**, siendo ***TeX* quien maneja el diseño**. Los documentos *LaTeX* (*.tex) se pueden abrir con cualquier editor de texto pues no tienen formato ni códigos ocultos o instrucciones binarias (*LaTeX*, 2020).

2.3. Servicios

DigitalOcean Es un proveedor de alojamiento en la nube que ofrece servicios computacionales a entidades comerciales para que puedan escalar implementando aplicaciones de *DigitalOcean* que se ejecutan en paralelo en múltiples servidores en la nube sin comprometer el rendimiento (*What is DigitalOcean and why should you host apps on it?*, s.f.).

La mayoría de proveedores de servicios en la nube ofrecen características avanzadas demasiado complicadas que comprometen la interfaz de usuario, contrariamente a la interfaz de *DigitalOcean* que es estética, funcional y sencilla de utilizar. Proporciona APIs simples que, después de generarlas se utilizan con herramientas estándar de *HTTP* como *Curl* para poder invocarlas. Sus servidores en la nube se encuentran en potentes máquinas de núcleo *Hex* con almacenamiento dedicado *SSD*, *ECC Ram* y *RAID*. Además, ofrece redes privadas entre máquinas virtuales para ejecutar *clusters* de bases de datos y sistemas distribuidos en algunas regiones. Posee una comunidad digital activa que ayuda respondiendo consultas. Los expertos siempre están allí para ayudar con las nuevas tecnologías de vanguardia (*What is DigitalOcean and why should you host apps on it?*, s.f.).

Uno de los servicios más usados para la autogestión son los *Droplets*. En esencia, son máquinas virtuales flexibles basadas en *Linux* que se ejecutan sobre un *hardware* virtualizado. Cada *Droplet* creado actúa como un nuevo servidor físico para el usuario (*What is DigitalOcean and why should you host apps on it?*, s.f.).

Docker Hub Es un **repositorio basado en la nube** en el que los usuarios y socios de Docker **crean, prueban, almacenan y distribuyen imágenes de contenedores**. A través de Docker Hub un usuario puede acceder a repositorios públicos de imágenes de código abierto y utilizar su espacio para crear

sus propios repositorios privados, funciones, compilaciones automatizadas y grupos de trabajo (*Working With Docker Hub*, s.f.).

Se pueden usar repositorios tanto públicos como privados. En los primeros los usuarios pueden compartir y colaborar con imágenes; en los segundos los datos están protegidos y son confidenciales. Es destacable aclarar que la versión gratuita de Docker Hub permite un repositorio privado (*What is Docker Hub? - Definition from WhatIs.com*, s.f.).

GitLab Es una **compañía global que proporciona alojamiento para control de versiones** de desarrollo de software utilizando *Git*. Es un servicio de alojamiento de repositorios *Git*, pero agrega muchas de sus características propias. Si bien *Git* es una herramienta de línea de comandos, *GitLab* proporciona una interfaz gráfica basada en la web (*What is GitLab?*, s.f.).

El sitio proporciona funciones similares a las redes sociales, como *feeds*, seguidores, *wikis* y un gráfico de red social para mostrar cómo los desarrolladores trabajan en sus versiones (*forks*) de un repositorio y qué bifurcación es la más nueva. Un usuario debe crear una cuenta para contribuir con el sitio, pero cualquier persona puede navegar y descargar repositorios públicos. Con una cuenta registrada se puede administrar repositorios, enviar contribuciones, revisar cambios de código, etc. Las tres características (*fork*, *pull request* y *merge*) son las que hacen que *GitLab* sea tan poderoso (*GitLab*, 2020).

¿DinaHosting? Se trata de una empresa que brinda alojamiento de datos a particulares, profesionales, empresas y organizaciones, así como alojamiento avanzado web y gestión de dominios. En este trabajo se ha utilizado como gestor del dominio adquirido donde se albergan todos los servicios (*DINAHOSTING / Dominios.es*, s.f.).

Capítulo 3

Desarrollo e Implementación

El presente trabajo fin de grado se ha desarrollado desde el punto de vista de un perfil *DevOps Developer*¹/*Cloud Architect*². Una de sus primeras atribuciones es la de crear un entorno aislado y completamente funcional para cada uno de los servicios de los que consta la aplicación *Cloud*. Esto se consigue mediante la tecnología de contenedores, configurando cada uno de ellos de tal manera que puedan comunicarse entre sí. También facilitaremos el proceso de creación y despliegue de los contenedores en local, de manera totalmente transparente para el resto del equipo involucrado en el desarrollo de la aplicación. Además ha de desarrollar el servicio encargado de brindar Integración Continua y Despliegue Continuo en los servidores, así como servicios para monitorizar la aplicación *Cloud*.

El escenario que se analiza es el de una empresa tecnológica involucrada en el desarrollo de una arquitectura para una aplicación *Cloud*. Para este fin, necesitamos recurrir a una aplicación ejemplo. Se ha optado por una aplicación ya existente, una aplicación de votación muy simple, pero que reúne las características que queremos ilustrar: consta de múltiples servicios interconectados que no sólo han de comunicarse entre sí sino hacerlo de manera segura.

Cabe destacar que los resultados trabajo son generalizables para otras empresas y aplicaciones, con el uso de diferentes lenguajes y servicios, ajustándonos a las necesidades del proyecto en cuestión.

3.1. Descripción de la aplicación ejemplo

Esta aplicación es la que está en desarrollo por la empresa de nuestro escenario. La aplicación elegida (*dockersamples/example-voting-app*, 2020), consta de cinco servicios que de manera esquemática se muestran en la Figura 3.1. Estos servicios son:

¹Es una práctica que tiene como objetivo la unión del desarrollo y la operación del software

²Encargado de diseñar la arquitectura que tendrá la aplicación para su despliegue final

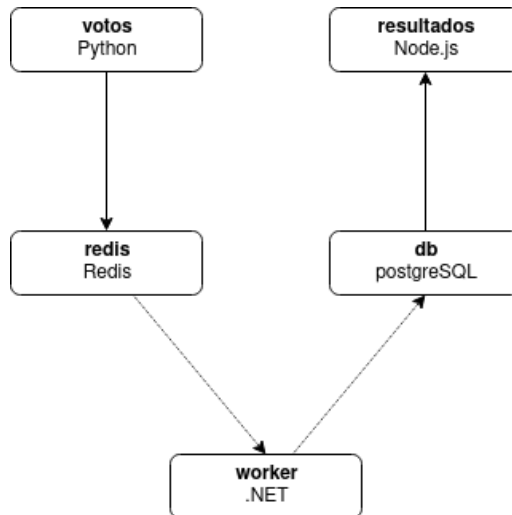


Figura 3.1: Arquitectura simplificada de la aplicación

votos Servicio escrito en *Python*, el cual ofrece al usuario la posibilidad de votar a *dog* o *cat* mediante dos botones. Este servicio, tal y como está implementado, se puede considerar que realiza las tareas de **front-end** y de un **back-end**.

resultados Servicio escrito en *Node.js*, el cual muestra los resultados actuales de la aplicación que se actualizan de manera constante, de tal manera que si se produce un voto, éste se ve reflejado inmediatamente.

redis Servicio que sirve como aliviador de carga para realizar los votos del servicio **votos**. Cuando se realiza un voto este se manda de manera asíncrona a la base de datos relacional **db** permitiendo que muchas personas voten a la vez almacenando en *RAM* las consultas por un periodo de tiempo en redis, hasta que el **worker**, consuma los votos restantes.

db Servicio que almacena los votos en una base de datos relacional manejada por *PostgreSQL* alojando los datos en el disco duro del sistema donde se encuentre.

worker Servicio escrito en *Java*, encargado de consumir los votos que **redis** procesa y de almacenarlos en la base de datos **db**

Este es un **breve resumen** de cómo es el **flujo de comunicación entre los servicios** vistos de la la aplicación:

1. Se realiza un voto en **votos** y este lo reenvía a **redis**
2. El voto³ es procesado y almacenado por **redis**
3. El servicio que se encarga de consumir el voto es el **worker**, escribiéndolo en la base de datos **db**.
4. Mientras tanto el servicio de **resultados** se encuentra consultando constantemente la **db** para mostrar un resultado.

³El voto se actualiza y almacena de manera asíncrona

3.1.1. Desarrollo tradicional frente a las nuevas tendencias

Es interesante mencionar que el desarrollo realizado en este trabajo fin de grado es fruto de la experiencia del autor en una de las empresas en las que ha trabajado. El escenario de partida fue un pequeño proyecto software que requería de manera urgente un escalado considerable, para ajustar la aplicación a las necesidades emergentes del mercado al que iba dirigida.

En aquel momento la empresa tenía un trabajador especializado en cada servicio desarrollando en un lenguaje específico. Los empleados instalaban todas las herramientas necesarias para trabajar su parte, junto con las herramientas necesarias para ofrecer el servicio en local y poder depurarlo, esto es, instalar las herramientas necesarias de los demás desarrolladores para poder ver la aplicación en conjunto y comprobar cómo se comportaban todas las partes juntas. Esto se traduce en mucha pérdida de tiempo instalando software, así como en la actualización de múltiples paquetes y dependencias del sistema que a veces no es del todo necesario.

En este contexto, nos planteamos ¿qué ocurriría si hubiera un nuevo integrante en el equipo? Habría que repetir el proceso de instalación de todas las herramientas de los demás integrantes del equipo, además de explicar la manera de testear la aplicación, lanzando servicios que al nuevo integrante, probablemente, no le incumbieran, etc.

Además de todos estos inconvenientes, la falta de un entorno de trabajo único para todos, es un problema crucial, ya que dificulta aspectos clave como el testeo de la aplicación y la depuración de errores (situación típica en la que la aplicación compila en un equipo pero no en otro).

Con la integración de un perfil especializado de *Cloud Architect* estos inconvenientes se palian en gran medida. La implantación de un sistema de trabajo mediante contenedores, junto con *scripts* de ayuda, permiten a los desarrolladores hacer un uso del sistema de manera más transparente y sin necesidad de tener conocimientos avanzados al respecto.

Además resolvemos todos los problemas mencionados anteriormente con el flujo de trabajo convencional:

- **Instalación de herramientas.** Los servicios se ejecutan en un entorno aislado gracias a los contenedores previamente desarrollados, por tanto, únicamente será necesaria la descarga de las imágenes Docker junto con la herramienta `docker`.
- **Pérdida de tiempo innecesaria.** El encargado de mantener el entorno en el cual se despliegan los servicios es del *DevOps*, por tanto la tarea de mantener actualizados los paquetes y dependencias de cada uno de los servicios compete únicamente a éste. Los demás desarrolladores solo deben preocuparse de tener la última imagen Docker en sus máquinas.

- **Automatización y fácil manejo.** El perfil *DevOps* tendrá como misión facilitar el uso de la herramienta **docker** a los demás trabajadores mediante *shell scripts*. De esta forma los programadores podrán, de manera independiente, desplegar, borrar y recompilar su código fuente en los contenedores de una manera sencilla.
- **Versionado.** Una de las mayores ventajas de trabajar con Docker son las imágenes que generamos; en ellas se encuentran todo el entorno diseñado para ejecutar un servicio en concreto (código fuente, paquetes, librerías, dependencias, etc). Podemos crear múltiples versiones y hacer uso de la plataforma *Docker Hub* para subir nuestras imágenes y disponer de ellas en cualquier máquina si nos hiciera falta.
- **Entorno único.** Gracias a la creación de imágenes y a que todos los miembros del equipo pueden descargarlas y usarlas, nos aseguramos que todos los entornos de trabajo son prácticamente idénticos y tendremos menos problemas a la hora de resolver incongruencias de compilación o ejecución de nuestra aplicación.
- **Escalado y alta disponibilidad.** Dada la facilidad con la que se puede desplegar un servicio utilizando contenedores, simplemente disponiendo de las imágenes de estos, la escalabilidad horizontal es un punto fuerte de estos. Podremos tener múltiples servidores y desplegar en ellos nuestros servicios según más nos convenga en función a la carga de trabajo que tengan. O bien podremos replicarlos para que, en caso de un fallo en uno de nuestros servidores, haya otro ejecutando los mismos servicios y no perdamos nunca la disponibilidad de nuestra aplicación al público.

3.1.2. Agentes interventores

En nuestra aplicación *Cloud* ejemplo harán falta varios perfiles que se encargarán de desarrollar código para sus respectivos servicios. Esta manera de estructurar la forma de trabajo nos da como resultado un equipo totalmente especializado en su materia pudiéndole sacar todo el potencial a cada uno de los integrantes, centrándolos en lo que mejor saben hacer.

- **Desarrolladores *Node.js*.** Encargado de modelar el servicio que ofrecerá el resultado de la votación. Asegurándose de que se realice de manera *responsive* y con un estilo agradable al usuario.
- **Desarrolladores *Python*.** Encargado de modelar el servicio que ofrecerá la página principal de votación. Tendrá que conocer cómo comunicarse con una base de datos volátil, además de implementar una interfaz amigable y sencilla para el usuario.

- **Desarrolladores *Java/.NET Framework*.** Encargado de la comunicación asíncrona que se produce entre la base de datos estática PostgreSQL y la volátil Redis.
- **Desarrollador de la arquitectura *DevOps*.** Encargado de realizar todo lo descrito en este trabajo, es decir, construir un entorno para todos los servicios, así como *scripts* de automatización, implementación de la metodología *CI/CD*, monitorización de los servidores, establecer seguridad en las comunicaciones y mantenimiento de los servicios una vez puestos en producción.

3.2. Descripción del entorno

A continuación describimos la estructura general del proyecto y destacamos algunos aspectos importantes de las decisiones de diseño adoptadas. En la Figura 3.2 se muestra una visión global de la estructura del proyecto.

Cada directorio que encontramos colgando del raíz forma parte de uno de los servicios a desarrollar para la implementación de la aplicación. Cada una de estas carpetas se encuentran subidas a *GitLab*. El proyecto completo se encuentra en un grupo llamado `marktfgr` y cada una de las carpetas se encuentran en subgrupos diferentes, es decir, en repositorios distintos, esto es una estrategia de organización útil para incorporar el *Despliegue Continuo* en el Capítulo 5.

Observamos que todos los directorios relacionados con los servicios que componen la aplicación, `TFG/ResultNone`, `TFG/VotePython` y `TFG/Worker`, poseen dos archivos comunes. Estos archivos son responsabilidad del perfil *DevOps* y definen la imagen del contenedor Docker para dicho servicio (`Dockerfile`) y la configuración requerida para desplegar dicha imagen generada (`docker-compose.yml`). No es estrictamente necesario que se ubiquen en el mismo directorio donde se encuentran los ficheros fuentes del servicio, pero si es una buena forma de estructurar el trabajo, además de ser más sencillo configurar las rutas internas de los contenedores si estos se encuentran en el directorio raíz del servicio.

Obsérvese que de los cinco servicios principales que consta la aplicación únicamente vemos tres directorios relacionados con los mismos, `TFG/ResultNone`, `TFG/VotePython` y `TFG/Worker`, esto se debe a que existen dependencias fuertes entre tres de los servicios existentes, concretamente entre `redis`, `db` y `worker`. Por tanto el despliegue y la generación de imágenes necesarias para dichos contenedores se realiza dentro del directorio `TFG/Worker`.

TFG/ Carpeta raíz donde albergaremos todos los ficheros del proyecto. Recordemos que usaremos *Git* para la gestión del versionado de archivos y crearemos el proyecto en *GitLab*, pues nos ofrece la posibilidad de crear ilimitados repositorios privados de manera gratuita.

Dada la estructura del directorio, tendremos que manejar cuatro repositorios a la

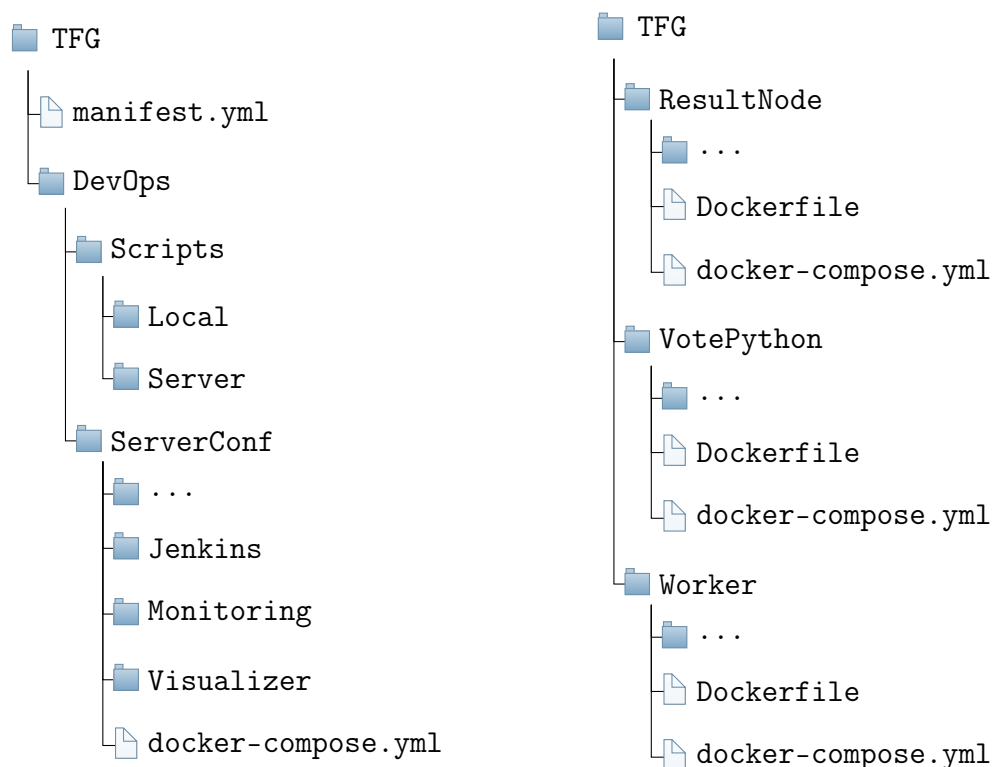


Figura 3.2: Estructura general del proyecto

vez; para facilitar esta tarea usaremos `tsrc` (Ver Anexo I:A.3), que nos permite conocer el estado de cada repositorio y automatizar comandos iterables en cada uno de estos. El archivo de configuración para ésta es `manifest.yml` donde se guarda toda la información para clonar los cuatro repositorios con los que trabajaremos.

Nota: Este directorio no contendrá control de versiones.

Como hemos mencionado antes, los directorios correspondientes a los servicios de la aplicación contienen el código fuente desarrollado por los miembros del grupo de trabajo responsable de los mismos junto con los ficheros del perfil *DevOps*; pasaremos a describir entonces la última carpeta:

- ***TFG/DevOps/*** El resto de ficheros relacionados con el perfil *DevOps* se encuentran, como su propio nombre indica, ubicado en este directorio.
 - ***TFG/DevOps/Scripts*** Ubicados todos los *scripts* necesarios en local para facilitar el control de los contenedores y como ayuda en el mantenimiento y actualización de las imágenes para el perfil *DevOps*. También encontramos algunos ficheros de configuración que veremos más adelante en el Capítulo 4.
 - ***TFG/DevOps/ServerConf*** Contiene la definición de los servicios para aplicar *CI/CD* y el monitorizado de los servidores, junto con la definición global para lanzar la aplicación completa en Docker Swarm. También encontraremos ficheros de configuración para los servicios.

Capítulo 4

Arquitectura Local

Este capítulo se centra en los ficheros generados por el *DevOps* y en la finalidad que tiene cada uno de estos, además resaltaremos fragmentos de código que se consideran más relevantes o importantes. Entraremos en detalle sobre la arquitectura en la que se ha organizado el proyecto y justificaremos estas decisiones. Estudiaremos cómo sería un flujo de trabajo en Local para hacernos una idea de las herramientas y *scripts* desarrollados junto con ejemplos de uso. Por último plantearemos cómo deben hacerse las labores de mantenimiento que la aplicación vaya requiriendo así como el enfoque y las pautas que habría que dar al proyecto en caso de requerir un nuevo servicio que integrar a lo ya existente.

4.1. Imágenes Docker

Los **Dockerfile** son archivos fundamentales para la utilización de contenedores Docker, pues es donde se definen las instrucciones para generar imágenes Docker. Como hemos descrito en la Sección 3.2, estos archivos se encuentran ubicados en cada uno de los directorios raíz del servicio, junto al código fuente de los mismos.

Describiremos brevemente las instrucciones más relevantes de los **Dockerfile** de cada uno de los servicios.

VotePython/Dockerfile (Figura 4.1) Este fichero es un buen ejemplo de lo que normalmente tendría un **Dockerfile**, pues es bastante simple.

ResultNode/Dockerfile (Figura 4.2) Destacamos que en este caso instalamos un paquete especial para node llamado *nodeamon*, el cual recompila la aplicación cuando detecta un cambio en nuestro código fuente.

Worker/Dockerfile (Figura 4.3) En este caso destacamos una técnica usada con frecuencia para reducir el tamaño de las imágenes generadas, llamada *multi-stage*. Esta técnica se aplica sobre todo cuando la fase de compilación de nuestro servicio genera ficheros estáticos o ficheros que por si solos nos sirven para

```

#Descargamos la imagen base oficial de Python
FROM python:2.7-alpine
#Configuramos directorio de la aplicación
WORKDIR /app
#Copiamos nuestro código a /app
COPY . .
#Instalamos los requisitos de la aplicación
RUN pip install -r requisitos.txt
#Exponemos el puerto 80 para servir la aplicación
EXPOSE 80
#Servimos la aplicación con 'gunicorn'
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--workers", "4", "--keep-alive", "0"]

```

Figura 4.1: Dockerfile para generar la imagen del servicio votos.

```

# Descargamos la imagen base oficial de Node
FROM node:10-slim
#Configuramos directorio de la aplicación
WORKDIR /usr/src/app
#Copiamos nuestro código a /usr/src/app
COPY . .
#Instalamos las dependencias del código
RUN npm install -g nodemon
RUN npm ci \
    && npm cache clean --force
#Asignamos el puerto 80 a la var. de entorno
ENV PORT 80
#Exponemos el puerto
EXPOSE 80
#Servimos la aplicación con 'nodeamon'
CMD ["nodemon", "server.js"]

```

Figura 4.2: Dockerfile para generar la imagen del servicio resultados.

```

#Primera fase donde descargamos la imagen base de Maven. Con esta imagen compilaremos el código
↪ fuente. Para poder referenciarla en la próxima fase usamos la opción 'AS' de 'FROM'
FROM maven:3.5-jdk-8-alpine AS constructor
WORKDIR /code
COPY pom.xml /code/pom.xml
RUN ["mvn", "dependency:resolve"]
RUN ["mvn", "verify"]
#Copiamos ficheros fuentes y creamos .jar
COPY ["src/main", "/code/src/main"]
RUN ["mvn", "package"]

##### MULTI-STAGE #####

#Descargamos la imagen con openjdk
FROM openjdk:8-jre-alpine
#Copiamos '.jar' creado en la fase anterior, para ello usamos la opción 'from' y le asignamos
↪ el nombre 'constructor'
COPY --from=build /code/target/worker-jar-with-dependencies.jar /
#Ejecutamos el servicio
CMD ["java", "-XX:+UnlockExperimentalVMOptions", "-XX:+UseCGroupMemoryLimitForHeap", "-jar",
↪ "/worker-jar-with-dependencies.jar"]

```

Figura 4.3: Dockerfile para generar la imagen del servicio **worker**.

ejecutar el servicio. Usamos una imagen inicial con las dependencias necesarias para compilar el código fuente y le agregamos un nombre para referirnos a ella posteriormente (imagen pesada). Una vez tenemos los ficheros compilados usamos otra imagen que tenga lo necesario para desplegar el servicio (imagen ligera) y copiamos de la construcción anterior los ficheros generados.

El resto de servicios como **redis** y **db** no contienen un fichero específico para definir la imagen, pues no se ha requerido ninguna modificación adicional a la ya ofrecida en el repositorio de imágenes Docker en *Docker Hub*.

4.2. Despliegue de contenedores Docker

Una vez tenemos las imágenes Docker generadas es hora de lanzar nuestros servicios en contenedores. Para ello podemos hacerlo directamente desde la terminal (ver más en la Subsección A.1.3) o, para agilizar el proceso, podemos definir en un fichero **docker-compose.yml** las reglas y la configuración que necesita nuestro contenedor.

Obsérvese que, a pesar de ser un archivo que define las instrucciones para el despliegue de imágenes, tenemos una opción **build** que corresponde a las configuraciones necesarias para construir la imagen. Esto es debido a que si no disponemos de la imagen Docker necesaria para lanzar el contenedor, mediante esta configuración **docker-compose** pasará a construir primero la imagen y después a desplegarla.

En este caso todos los servicios tienen al menos una **red externa** definida. Este tipo de redes han de ser creadas manualmente con el comando **docker network create**, antes de ejecutar el fichero **docker-compose.yml** ya que la herramienta no puede configurarlas. En caso de que la red no fuera externa, no sería necesaria la creación

```

#Versión del formato escogido
version: '3.1'
services: #Definición de servicios
  vote-web: #Nombre del servicio
    build: #Configuraciones para la creación de imagen
      context: .
      dockerfile: Dockerfile
    container_name: tfg-frontend-vote #Nombre contenedor
    image: votepython_frontend:latest #Imagen necesaria
    ports: #Mapeo de puerto
      - "5000:80"
    networks: #Redes usadas
      - frontend
    volumes: #Mapeo de volúmenes
      - './app'
networks: #Definición de redes a usar
  frontend: #Nombre interno de la red para compose
    external: #Indica que debe existir previamente
      name: tfg-frontend #Nombre de la red externo

```

Figura 4.4: docker-compose.yml para el servicio votos.

previa a la ejecución del fichero, pues éste la crea por nosotros. Podemos observar que en la definición de las redes usaremos un nombre para referirnos a ellas dentro del docker-compose.yml (frontend y backend), que no tiene por qué ser igual que la red que usaremos fuera en los contenedores (tfg-frontend y tfg-backend).

Es muy importante que el programador pueda seguir depurando el código, probar fallos y testear la aplicación en el desarrollo local, sin tener que estar reconstruyendo el contenedor constantemente, es por ello que se aprovecha la posibilidad de mapear directorios del *host* en el contenedor que nos interese. De este modo podemos utilizar servicios como **nodeamon** visto en Figura 4.2.

Describiremos brevemente las instrucciones más relevantes de los docker-compose.yml de cada uno de los servicios. Para más información sobre el **mapeo de puertos** o **volúmenes** véase en la Subsección A.1.3.

VotePython/docker-compose.yml (Figura 4.4) Este es un ejemplo básico de docker-compose.yml.

ResultNode/docker-compose.yml (Figura 4.5) Si queremos conocer la diferencia entre los dos volúmenes creados en este fichero docker-compose.yml se habla de ello en la Subsección A.1.3.

Worker/docker-compose.yml (Figura 4.6) Ejemplo de como lanzar múltiples servicios (redis, db y worker) definidos en un único fichero docker-compose.yml.

```

version: '3.1'
services:
  result-web:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: tfg-frontend-result
    image: resultnode_frontend:latest
    ports:
      - "5001:80"
    networks:
      - backend
    volumes: #Mapeo y Definición de volúmenes
      - './usr/src/app'
      - TFG-NodeResult:/usr/src/app/node_modules
networks:
  backend:
    external:
      name: tfg-backend
volumes: #Definición del volumen
  TFG-NodeResult:

```

Figura 4.5: docker-compose.yml para el servicio resultados.

```

version: '3.1'
services: #Definición de 3 servicios
  worker: #Primer servicio
    build:
      context: .
      dockerfile: Dockerfile
    container_name: tfg-backend-worker
    image: worker_backend:latest
    depends_on: #No lanzar el servicio 'worker' hasta haber desplegado 'redis' y 'database'
      - "redis"
      - "database"
    networks:
      - backend
      - frontend
  redis: #Segundo servicio
    image: redis:alpine
    container_name: tfg-backend-redis
    ports: ["6379"]
    volumes:
      - "TFG-redisDB:/data"
    networks:
      - frontend
  database: #Tercer servicio
    image: postgres:9.4
    container_name: tfg-backend-database
    volumes:
      - "TFG-postgresDB:/var/lib/postgresql/data"
    networks:
      - backend
volumes:
  TFG-postgresDB:
  TFG-redisDB:
networks:
  backend:
    external:
      name: tfg-backend
  frontend:
    external:
      name: tfg-frontend

```

Figura 4.6: docker-compose.yml para el servicio worker.

4.3. Scripts

Recomendamos leer previamente la Subsección A.1.3 donde se explican algunos comandos básicos para manejar `docker` y `docker-compose`.

Con la finalidad de centrarnos en las partes más importantes de los *scripts*, se han omitido algunas partes del código o reescrito (mensajes por pantalla) como comentarios, u omisión de funciones similares a las ya mostradas con anterioridad, a fin de facilitar la comprensión de éstos.

rebuildDeploy.sh (Figura 4.7) La función de este *script* es construir la imagen Docker del servicio que se le pase como argumento mediante las definiciones descritas en el archivo `Dockerfile` (Visto en la Sección 4.1) y desplegarlos usando las directivas del fichero `docker-compose.yml` (Visto en la Sección 4.2) que se encuentran alojados en el directorio raíz de dicho servicio.

OPCIONES	DESCRIPCIÓN
<code>all</code>	<i>Despliega todos los servicios involucrados en la aplicación</i>
<code>backend</code>	<i>Despliega los tres servicios relacionados a este entorno, es decir, redis, worker y postgresSQL</i>
<code>frontendResult</code>	<i>Despliega únicamente el servicio Node.js encargado de mostrar los resultados</i>
<code>frontendVote</code>	<i>Despliega únicamente el servicio de Python encargado de mostrar las opciones de votación</i>

Tabla 4.1: Opciones disponibles y breve descripción

Podemos ver el resultado final tras ejecutar el script:

```
[mark@Tilted-PC ~]$ docker ps
```

CONTAINER ID	IMAGE	PORTS	NAMES
b297d480cf43	resultnode_frontend:latest	0.0.0.0:5001->80/tcp	tfg-frontend-result
8e2b45fe05a2	votepython_frontend:latest	0.0.0.0:5000->80/tcp	tfg-frontend-vote
b828d07f315a	worker_backend:latest		tfg-backend-worker
4f08316bea69	redis:alpine	0.0.0.0:32768->6379/tcp	tfg-backend-redis
996f6acd9d0b	postgres:9.4	5432/tcp	tfg-backend-database

downDeploy.sh (Figura 4.8) Script para detener uno, varios o todos los servicios que componen la aplicación. Las opciones de este script son las mismas que vimos para *rebuildDeploy.sh* en la Tabla 4.1

cleanVotes.sh (Figura 4.9) Nos permite eliminar los votos de la base de datos. Con este script contemplamos todos los estados posibles en los que se puede encontrar la aplicación antes de borrar dichos votos. *Nota: este script carece de argumentos.*

```
#!/bin/bash

#Definimos PATH de los ficheros para realizar el despliegue
dir_vote_frontend="${MARKTFG}/VotePython/docker-compose.yml"
dir_result_frontend="${MARKTFG}/ResultNode/docker-compose.yml"
dir_worker_backend="${MARKTFG}/WorkerJava/docker-compose.yml"

#Función para crear las redes externas
function createNetworks(){
    if $(docker network ls | grep tfg | awk {'print($2)'})
    then
        docker network create tfg-frontend
        docker network create tfg-backend
    else
        #Mensaje de error, redes existentes
    fi
}

#Función que construye la imagen y lanza el servicio
function frontend_Vote(){
    docker-compose -f $dir_vote_frontend build
    docker-compose -f $dir_vote_frontend up -d
}

function frontend_Result(){ ... }
function backend(){ ... }

#Creamos la red
createNetworks
#Ejemplo en caso de elegir la opción "all"
case $1 in
    all)
        #Llamamos a todas las funciones en paralelo
        frontend_Vote &
        frontend_Result &
        backend &
        wait
        frontendVote) ...
        frontendResult) ...
        backend) ...
        *) ...
esac
```

Figura 4.7: Script rebuildDeploy.sh para compilar imágenes y desplegarlas

```
#!/bin/bash

#PATH ficheros con los cuales se desplegaron
dir_vote_frontend="${MARKTFG}/VotePython/docker-compose.yml"
dir_result_frontend="${MARKTFG}/ResultNode/docker-compose.yml"
dir_worker_backend="${MARKTFG}/WorkerJava/docker-compose.yml"

#Funcion para parar servicio
function frontend_Vote(){
    docker-compose -f $dir_vote_frontend down
}
function frontend_Result(){ ... }
function backend(){ ... }

#Limpiar el sistema de contenedores e imágenes 'muertas'
function cleanDocker(){
    docker system prune -f
}
#Ejemplo en caso de elegir la opción "all"
case $1 in
    all)
        frontend_Vote &
        frontend_Result &
        backend &
        wait
        cleanDocker
        ;;
    frontendVote) ...
    frontendResult) ...
    backend) ...
    *) ...
esac
```

Figura 4.8: Script downDeploy.sh para compilar imágenes y desplegarlas


```
#!/bin/bash

FILES="${MARKTFG}/DevOps/Scripts/Local"$

#Comprobamos si existen volúmenes relacionados con el TFG
if (docker volume ls -qf name=_TFG- | grep -i tfg > /dev/null); then
    ...
else
    exit 1
fi

#Comprobamos si existen servicios del tfg en ejecución
if ( docker ps -f name=tfg-* --format "table {{.Names}}\t" | tail -n +2 | grep -i tfg >
↪ /dev/null); then
    #Confirmación del borrado
    echo -ne "[?] Desea borrar la base de datos ?"; read ANS
    if (echo "${ANS}" | grep -i '^y'); then
        #Paramos y eliminamos los contenedores
        $FILES/downDeploy.sh all
        #Limpiamos la base de datos (Eliminamos volúmenes)
        docker volume ls -qf name=_TFG- | xargs -r docker volume rm
        echo -ne "[?] Desea relanzar todos servicios ?"; read ANS
        if (echo "${ANS}" | grep -i '^y'); then
            #Lanzamos la aplicación completa
            $FILES/rebuildDeploy.sh all
            #Acabamos. Servicio desplegado
        else
            #Acabamos. Sin servicio desplegado
        fi
    else
        #Acabamos. Ningún cambio realizado
    fi
fi

#Si no existían servicios del tfg en ejecución repetimos pero sin detener previamente
else
    ...
fi
```

Figura 4.9: Script `cleanVotes.sh` para limpiar la base de datos de los votos realizados

push-images.sh Script que nos facilita la tarea de publicar las imágenes al repositorio de *Docker Hub*, automatizando el proceso de compilación, *tag* de imágenes, registro de cambios y limpieza del entorno una vez subidas las imágenes.

El proceso de subida de una imagen comienza con la definición de variables necesarias (ver Figura 4.10). Una vez tenemos las variables configuradas pasamos a la selección del servicio que deseamos subir (ver Figura 4.11), llamando a la función de cada servicio en la cual realizaremos diferentes tareas. Estas tareas pueden ser construir las imágenes, agregarles un *tag* para poder subirlas al repositorio (ver Figura 4.12), subirlas (ver Figura 4.13) y por último en caso de tratarse de una imagen para producción agregar un mensaje de la versión y limpiar el entorno Docker local (ver Figura 4.14).

Nota: Este script requiere de dos argumentos que podremos ver cuales son y para que sirven en las tablas Tabla 4.2 y Tabla 4.3

OPCIONES \$1	DESCRIPCIÓN
dev	Especificamos el repositorio de Docker Hub perteneciente al entorno de trabajo de desarrollo todos los servicios involucrados en la aplicación
prod	Especificamos el repositorio de Docker Hub perteneciente al entorno de trabajo de producción

Tabla 4.2: Opciones disponibles primer argumento \$1 y breve descripción

OPCIONES \$2	DESCRIPCIÓN
worker	Subiremos la imagen correspondiente al servicio worker al repositorio dev prod
frontendResult	Subiremos la imagen correspondiente al servicio resultados al repositorio dev prod
frontendVote	Subiremos la imagen correspondiente al servicio votos al repositorio dev prod

Tabla 4.3: Opciones disponibles segundo argumento \$2 y breve descripción

script-completion.bash (Figura 4.15) Para poder usar de manera rápida y cómoda los *scripts*, se han definido unos **alias** a los mismos, con el fin de invocarlos en la terminal independientemente del directorio donde nos encontremos. Utilizamos estos **alias** para agregar la función de **autocompletar** las opciones que nos permiten nuestros *scripts*. Para poder verlo, deberemos escribir el nombre del alias asociado al *script* que estamos interesados usar y pulsar el **Tabulador** dos veces (como vemos en la Figura 4.16).

ssh.conf Este fichero se detalla en la Sección B.2. Se trata de una configuración que no es estrictamente necesaria, pero facilita la utilización de los scripts mencionados anteriormente.

.bashrc.conf Este fichero se describe en la Sección B.1. Nos facilita el acceso a los servidores mediante nombres impuestos.

```

##### VARIABLES #####
SERVER=$1
IMAGE=$2
ID="tfgdenis"
LOGS="${MARKTFG}/DevOps/Scripts/Local"
##### FIN VARIABLES #####

#Falta el primer argumento (dev/prod)
if [ ! $SERVER ] ; then
    exit 2
fi
#Falta el segundo argumento
if [ ! $IMAGE ] ; then
    exit 2
fi

# Directorio de los ficheros docker-compose.yml
dir_resultnode="${MARKTFG}/ResultNode/docker-compose-server.yml"
...
# TAG por defecto para las imágenes construidas en local
resultnode_local="resultnode_frontend:latest"
...
case $SERVER in
    dev)
        # TAGs para subir al repositorio de Docker-Hub
        resultnode_server="${ID}/dev_resultnode:"
        ...
    ;;
    prod)
        # Registros con info sobre imágenes subidas a Docker-Hub
        log_resultnode="${LOGS}/resultnode.log"
        ...
        # TAGs para subir al repositorio de Docker-Hub
        resultnode_server="${ID}/prod_resultnode:"
        ...
    ;;
esac

```

Figura 4.10: Script pushImages.sh. Definición de variables

```

#Subimos imagen deseada
function resultnode(){
    docker-compose -f $dir_resultnode build --no-cache
    tagImage result
    pushImage result
    if echo "${SERVER}" | grep -i "^prod"; then
        logImage result
    fi
    deleteImage result
}
function votepython(){ ... }
function worker(){ ... }

case $IMAGE$ in
    resultnode)
        resultnode
    votepython)
        worker)
        *)
esac

```

Figura 4.11: Script pushImages.sh. Selección de servicio a subir

```

#Función para agregar un tag a la imagen
function tagImage(){
    echo -ne "[!] Escribir tag (yyyymmdd):"
    read TAG
    case $1 in
        result)
            docker tag ${resultnode_local} ${resultnode_server}${TAG}
            docker tag ${resultnode_local} ${resultnode_server}latest
        ;;
        vote) ...
        worker) ...
    esac
}

```

Figura 4.12: Script `push-images.sh` Función para agregar TAGs a las imágenes

```

#Funcion para publicar imagen
function pushImage(){
    echo -ne "Subir imagen a Docker Hub? (y/n):"
    read RESP_PULL
    if echo "${RESP_PULL}" | grep -i "^y"; then
        case $1 in
            result)
                echo -ne "[?] Subir tag: LATEST?(y/n):"
                read RESP_TAG
                if echo "${RESP_TAG}" | grep -i "^y"; then
                    docker push ${resultnode_server}$TAG
                    docker push ${resultnode_server}latest
                else
                    docker push ${resultnode_server}$TAG
                fi
            ;;
            vote) ...
            worker) ...
        esac
    fi
}

```

Figura 4.13: Script `pushImages.sh` Función para subir las imágenes

```

#Función para guardar un 'commit'
function logImage(){
    if echo "${RESP_PULL}" | grep -i "~y"; then
        echo -ne "Commit Imagen: \e[0m"
        read MESSAGE
        case $1 in
            result)
                echo "[Tag: ${TAG}] - ${MESSAGE}" >> $log_resultnode
                ;;
            vote) ..
            worker) ...
        esac
    fi
}
#Función para limpiar
function deleteImage(){
    echo -ne "Borrar iamgenes generadas?(y/n):"
    read RESP_DEL
    if echo "${RESP_DEL}" | grep -i "~y"; then
        docker system prune -f
        case $1 in
            result)
                docker rmi ${resultnode_server}$TAG ${resultnode_server}latest
                ↪ ${resultnode_local}
                ;;
            vote) ..
            worker) ...
        esac
    fi
    docker system prune -f
}

```

Figura 4.14: Script pushImages.sh. Función para agregar, registrar cambios y limpiar entorno

```

#!/usr/bin/env bash
_push_images_completions()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    if [ $COMP_CWORD -eq 1 ]; then
        COMPREPLY=( $(compgen -W "dev prod" -- $cur) )
    elif [ $COMP_CWORD -eq 2 ]; then
        COMPREPLY=( $(compgen -W "resultnode votepython worker" -- $cur) )
    fi
    return 0
}
#Autocompletado de dos argumentos
complete -F _push_images_completions tfg-pushImages
#Autocompletado sencillo de un solo argumento
complete -W "all frontendVote frontendResult backend" tfg-rebuild tfg-down

```

Figura 4.15: Script rebuildDeploy.sh para compilar imágenes y desplegarlas

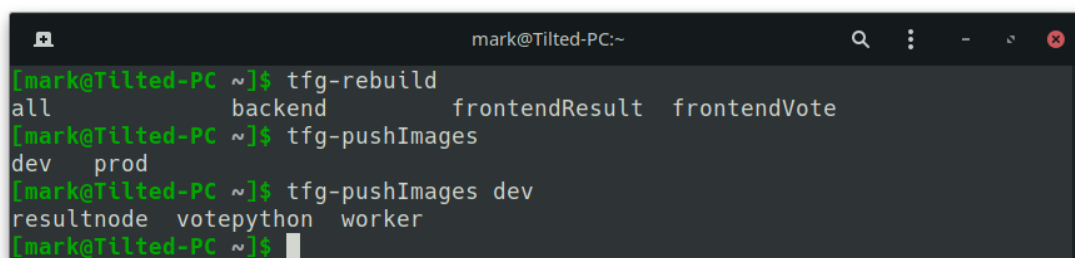
A terminal window titled 'mark@Tilted-PC:~' showing the execution of two commands. The first command is 'tfg-rebuild', followed by a tab completion showing 'all', 'backend', 'frontendResult', and 'frontendVote'. The second command is 'tfg-pushImages', followed by a tab completion showing 'dev' and 'prod'. The third command is 'tfg-pushImages dev', followed by a tab completion showing 'resultnode', 'votepython', and 'worker'. The prompt is '[mark@Tilted-PC ~]\$'.

Figura 4.16: Autocompletado tras pulsar dos veces el Tabulador en `tfg-rebuild` y `tfg-pushImages`

4.3.1. Configuraciones

Dado que trabajamos en local, no es necesario aplicar ninguna configuración extra al despliegue de los diferentes servicios, tales como configuraciones para mejorar la seguridad, conexión entre diferentes nodos, etc.

4.4. Infraestructura de la red

Ahora que sabemos como desplegar cada uno de los servicios que la aplicación requiere gracias a los scripts y hemos visto en la *Sección 4.2* la definición de cómo lanzarlos, vamos a estudiar la conexión de los contenedores unos con otros mediante las redes descritas. Figura 4.17

tfg-frontend En esta red encontraremos alojados los servicios de `votos`, `redis` y `worker`. En realidad, dado que el servicio encargado de pedir los votos contiene integrados la interfaz final que ve el usuario y el proceso de envío de los datos a la base de datos, es un servicio híbrido, pero hemos optado por colocarlo en esta capa.

tfg-backend En esta red encontramos la base de datos y el servicio encargado de mostrar el resultado.

El servicio de `votos` mapea el puerto 80 por el que sirve el proceso `nodemon` al 5000. Esto se hace por una cuestión de compatibilidad con otros posibles servicios que pudieran estar ejecutándose en el ordenar que despliega el servicio, pues se trata de un puerto muy común. Por otro lado pasa lo mismo con el proceso de `gunicorn` que es utilizado en el servicio de `resultados` donde se sirve por el puerto 80 pero lo mapeamos al 5001. La facilidad con la que podemos mapear los puertos en los contenedores nos ahorra mucho trabajo de configuración que tendríamos que realizar en los diferentes servicios para que no hubiera conflicto entre ellos por utilizar los mismos puertos por defecto.

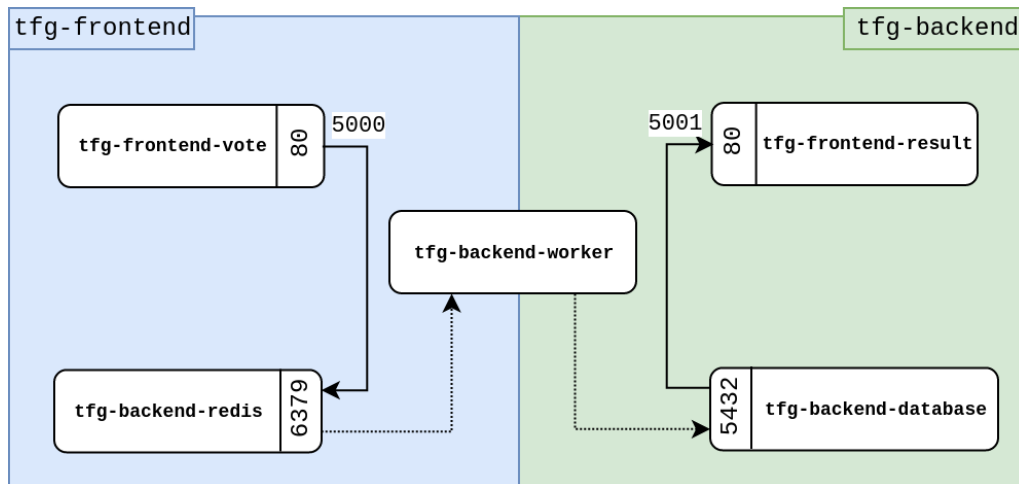


Figura 4.17: Infraestructura de red en el despliegue local

4.5. Flujo de trabajo en Local

Como mencionamos anteriormente, gracias a la configuración que tienen nuestros contenedores, los integrantes del equipo podrán trabajar sobre éstos, como si lo estuvieran haciendo en su maquina local, pues poseen todo lo necesario para la construcción y despliegue de sus servicios, además de contar con las facilidades para utilizar auto-compilación tras un cambio cualquiera en el código fuente.

4.6. Mantenimiento y nuevo desarrollo

El **mantenimiento de los servicios** en este caso consiste en la actualización de los paquetes y dependencias que los sistemas creados en los contenedores requieran, tales como actualizaciones de seguridad, corrección de fallos, etc.

También es posible que se quiera actualizar la versión de lenguaje que se esta usando, por ejemplo, si queremos actualizar el servicio de votación que está basado en *Python 2.7* a una versión más nueva, tendremos que realizar los cambios pertinentes en nuestro código fuente, así como cambiar el entorno en el cual se despliega, es decir, cambiando las reglas para la creación de la imagen Docker, empezando por modificar la versión de la imagen base. Vemos entonces, cómo con este pequeño cambio en Docker, hemos logrado el paso de una versión a otra sin complicaciones de corrupción de paquetes ni errores por conflicto o incongruencias en nuestros paquetes del sistema.

Para un caso en el que haya que agregar una nueva funcionalidad a nuestra aplicación a modo de servicio, se recomendará seguir la estructura, es decir, crear un repositorio específico para dicha funcionalidad, agregar los archivos Docker necesarios junto al código fuente del servicio y actualizar los *scripts* para contemplar un nuevo contenedor y una nueva imagen en nuestro entorno de desarrollo.

Capítulo 5

Arquitectura Cloud

Ahora que tenemos el entorno de desarrollo configurado y en marcha, es hora de subir un escalón y pasar al mundo *Cloud*.

En esta sección abordaremos los diferentes problemas y retos que encontramos al realizar un despliegue correcto en la nube. Veremos las diferencias y similitudes que hay que tener en cuenta respecto al entorno local con los contenedores y las imágenes de los mismos. Estudiaremos los nuevos *scripts* que el perfil DevOps debe tener para hacer más cómodo y eficiente su trabajo, así como las diferentes configuraciones para los servicios a implantar. Entraremos en detalle en la infraestructura de red, la seguridad de la misma y el nuevo flujo de trabajo que supone desarrollar en la plataforma Cloud. También explicaremos cómo trabajar de esta forma supone la capacidad para escalar de manera horizontal y aprovechar así la alta disponibilidad de nuestro servicio, además de la Integración Continua / Despliegue Continuo. Para finalizar trataremos cómo deben hacerse las labores de mantenimiento de los contenedores y los servidores y de las pautas a seguir en caso de incorporar más servicios a nuestra aplicación.

5.1. Nueva estructura de la aplicación

Dado que el entorno no es el mismo, la estructura de nuestra aplicación ha variado con respecto a la vista anteriormente en la Figura 3.1. Ahora usaremos un servicio más, como vemos en la Figura 5.1, que nos permite abrir la aplicación al público y que éste pueda acceder desde cualquier plataforma introduciendo nuestro dominio donde se encontrará alojada la aplicación. Para conocer la configuración del dominio véase la Sección C.4.

Este nuevo servicio es:

Ingress Se trata de un nuevo servicio necesario para comunicar nuestra aplicación alojada en los servidores al mundo exterior. Este servicio también actúa

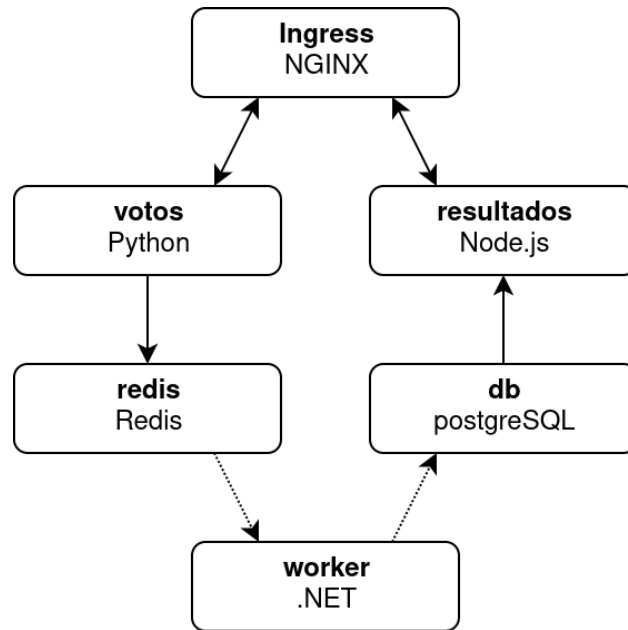


Figura 5.1: Arquitectura simplificada de la aplicación

como **reverse-proxy**¹ y balanceador de carga entre los diferentes servicios. Veremos más detalles sobre su implementación en la Subsección 5.3.1 y de la comunicación con el resto de la aplicación en Sección 5.7.

5.2. Imágenes Docker

Para usar las imágenes Docker en el servidor de manera correcta, debemos modificar los puertos usados por los microservicios, y reducir, si es posible, el tamaño de las imágenes para lograr mayor eficiencia en la compilación. A continuación repasaremos las modificaciones realizadas a los **Dockerfile** de nuestra aplicación, únicamente comentaremos las partes de código modificadas.

*Nota: La imagen del servicio **worker** no necesitará ninguna modificación, por tanto usaremos la misma que usamos para local junto con las imágenes base proporcionadas por la comunidad de Docker Hub para **redis** y **postgreSQL** (ver Figura 4.3).*

VotePython/Dockerfile-server (Figura 5.2) Esta imagen es la más sencilla de modificar, pues únicamente cambiaremos el puerto a exponer y configuraremos el servicio *unicorn* para que utilice ese puerto.

ResultNode/Dockerfile-server (Figura 5.3) En este caso, como mencionamos anteriormente es conveniente reducir el tamaño de nuestras imágenes, es por eso que aplicaremos la técnica de *multi-stage* que mencionamos con anterioridad en la Figura 4.3

¹Es un tipo de servidor proxy que recoge peticiones de un cliente desde uno o más servidores

DevOps/ServerConf/Jenkins/Dockerfile (Figura 5.4) Esta nueva imagen se usará para lanzar nuestro servicio **jenkins** (que detallamos en la Subsección 5.3.4) encargado de la Integración Continua / Despliegue Continuo. Existe una peculiaridad con esta imagen: con ella construiremos y lanzaremos contenedores y por esta razón requiere la instalación del paquete **docker-ce**. Además veremos que utilizamos la opción **USER** que nos permite seleccionar o crear un usuario y ejecutar todo lo que viene a continuación como dicho usuario.

5.3. Descripción de servicios

Comenzamos hablando de nuestra aplicación como un conjunto de servicios, que se desarrollan y se sirven de manera independiente mediante contenedores. En el mundo *Cloud* **nuestra aplicación será por si misma un servicio** (**tfg-denis**) y pasará a estar formada por **microservicios**. De esta manera nos encontramos con una nueva estructura. Un servidor puede servir múltiples servicios compuestos a su vez por dos o más microservicios.

Hemos usado **DigitalOcean** como servicio proveedor de servidores. Nuestro servidor estará formado por múltiples *Droplets*². De esta manera, la aplicación se podrá desplegar en máquinas separadas/aisladas como si se encontrara en una única, de forma análoga a como lo hacíamos en local. Detallaremos cómo conseguir esto en la Sección 5.5.

Describimos a continuación cómo hemos abordado los diferentes servicios en los servidores de *DigitalOcean*.

5.3.1. Nuestra aplicación como servicio - tfgdenis

Ahora nuestra aplicación ha pasado a ser un servicio. A pesar de que disponemos de hasta 3 nodos para desplegar nuestra aplicación, la arquitectura sigue siendo la misma, nada ha cambiado. En la Figura 5.5 vemos de que todas las aplicaciones se ejecutan bajo un contenedor y que, por consiguiente, podremos tener múlti-

²Servicio ofrecido por *DigitalOcean* para servidores de propósito estándar

```
FROM python:2.7-alpine
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
# Exponemos el puerto 5000
EXPOSE 5000
# Configuramos el puerto que expusimos anteriormente
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:5000", "--workers", "4", "--keep-alive", "0"]
```

Figura 5.2: Script VotePython Dockerfile Definición de reglas para la imagen

```

# Primera fase donde descargamos la imagen base de Node. Con esta imagen compilaremos el código
↳ fuente. Para poder referenciarla en la próxima fase usamos la opción 'AS' de 'FROM'
FROM node:10-slim AS constructor
WORKDIR /usr/src/app
COPY . .
#Instalamos las dependencias del código
RUN npm install
RUN npm ci && npm cache clean --force

##### MULTI-STAGE #####

FROM node:10-slim
WORKDIR /usr/src/app
#Asignamos el puerto 5001 a la variable de entorno
ENV PORT 5001
#Exponemos el puerto
EXPOSE 5001
#Copiamos ficheros creados en la fase anterior, para ello usamos la opción 'from' y le
↳ asignamos el nombre 'constructor'
COPY --from=build-stage /usr/src/app .
#Esta vez usamos 'node', en lugar de 'nodeamon'
CMD [ "node", "/usr/src/app/server.js" ]

```

Figura 5.3: Script ResultNode Dockerfile. Definición de reglas para la imagen

ples instancias. Éstas no tendrán por qué estar en la misma máquina física, si no que podrán encontrarse en nodos diferentes, conservando la comunicación gracias a *Docker Swarm* como veremos en la Sección 5.5. Tendremos en cuenta que necesitamos como microservicio extra el **ingress** que consistirá en una imagen base *Docker Hub* pre-construida que nos sirve un *reverse-proxy* con la automatización de *Let's Encrypt* y *Cerbot* para la creación y aplicación de certificados *SSL* de nuestro dominio (ver sección 5.8 sobre seguridad).

5.3.2. Servicio - monitoring

Este servicio se encargará de registrar todos los eventos que ocurran en el servidor. Dispondremos de microservicios en cada uno de los nodos encargados de auditar y extraer métricas, así como una interfaz muy visual y fácil de manejar.

Para llevar esta tarea a cabo, disponemos de un total de cuatro microservicios como podemos ver en la siguiente Figura 5.6.

node-exporter Es el encargado de monitorizar y registrar las métricas del nodo en el que se encuentra. Dado que se trata de una tarea que requiere comunicación directa con el *Kernel* debemos darle permisos suficientes para que funcione correctamente, en este caso deberemos configurarlo para cada uno de los nodos con los permisos de tipo 2 (Ver la Figura 5.7):

cadvisor Al igual que el microservicio anterior, éste se encarga de recoger métricas del nodo en el que se encuentra, pero específicamente de los contenedores que están desplegados sobre él.

```

#Descargamos imagen base
FROM jenkins/jenkins:lts
#Trabajamos como usuarios root
USER root
#Instalamos todos los paquetes que requiere docker-ce
RUN apt-get update && \
    apt -y full-upgrade && \
    apt-get -y install apt-transport-https \
        ca-certificates \
        curl \
        gnupg-agent \
        software-properties-common \
        pssh && \
# Agregamos el repositorio oficial de Docker e instalamos docker-ce
    curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg >
    ↪ /tmp/dkey; apt-key add /tmp/dkey && \
    add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") \
    $(lsb_release -cs) \
    stable" && \
    apt-get update && \
    apt-get -y install docker-ce
#Agregamos al grupo docker el usuario jenkins
RUN usermod -aG docker jenkins
#Finalizamos quedándonos como usuario jenkins
USER jenkins

```

Figura 5.4: Script Jenkins Dockerfile. Definición de reglas para la imagen

prometheus Es una solución *open-source* que nos facilita una herramienta muy potente para medir y auditar las métricas de los nodos, así como generación de alertas y muchas más herramientas de filtrado y visualización de datos.

Su interfaz es muy técnica y por ello no utilizamos el visualizador de esta herramienta, si no que lo haremos con Grafana. Además, dispone de un módulo, mencionado anteriormente (node-exporter) del que lee las métricas.

grafana Podremos realizar consultas a las métricas que nos ofrece Prometheus, y utilizar sus herramientas de filtrado para obtener y visualizar diferentes gráficas, así como porcentajes de uso o tiempo de ejecución. El panel que podemos ver en la *Figura 5.8* ha sido creado a medida para este servidor, existen plantillas que nos ayudan a personalizar nuestra visualización de métricas como mejor se adapte a nuestras necesidades. Se encuentra en formato *JSON* en TFG/DevOps/Monitoring/Dashboards/TFG-Metrics.json.

5.3.3. Servicio - visualizer

Se trata de un servicio ya estructurado y montado. Sirve para ver de manera visual los diferentes nodos que componen nuestro *cluster* y los distintos contenedores que se encuentran en ejecución en cada uno de éstos. Podemos ver un ejemplo en la *Figura 5.9* donde se encuentra desplegada nuestra aplicación con todos sus microservicios.

Nota: Observamos que el propio servicio también aparece en la Figura 5.9

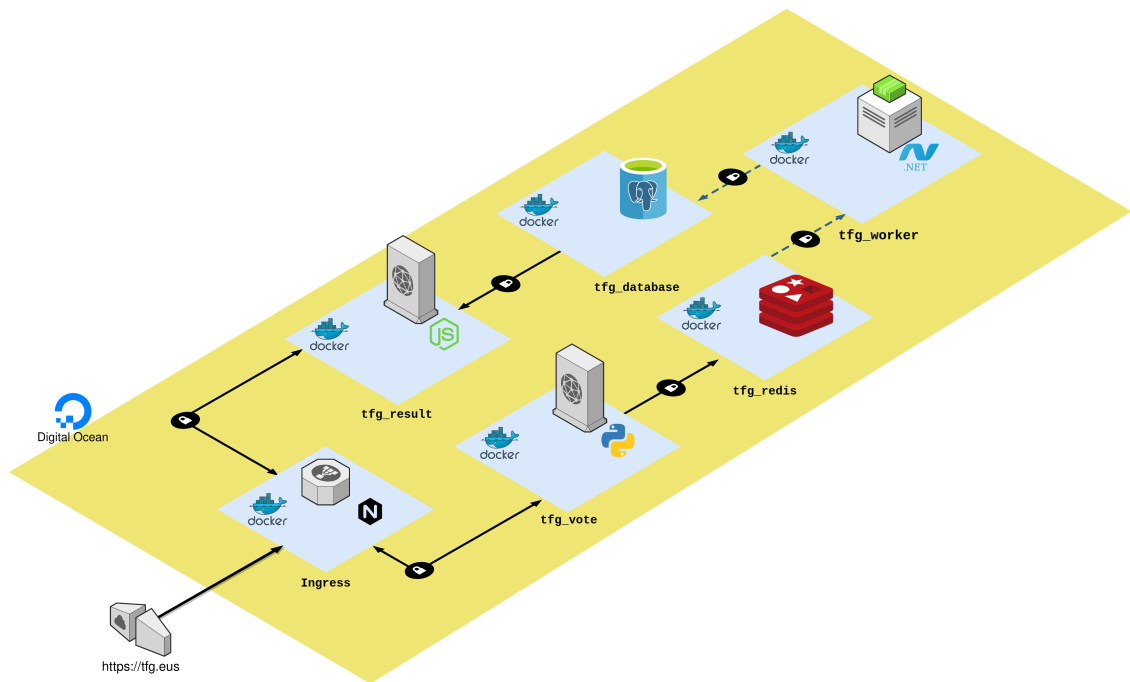


Figura 5.5: Arquitectura Cloud de la aplicación – Servicio tfgdenis

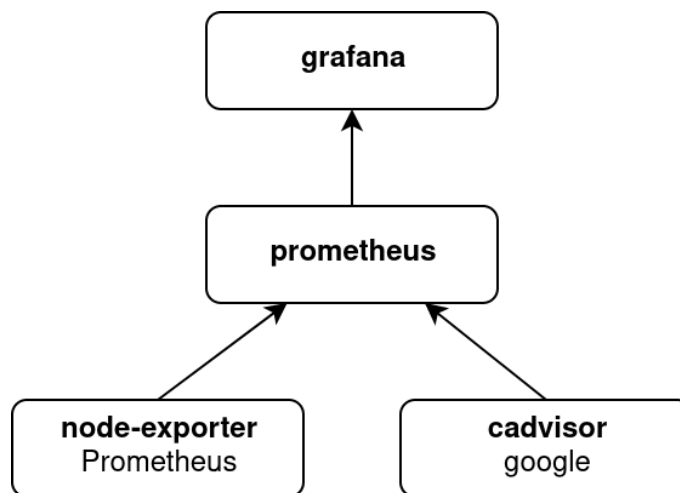


Figura 5.6: Arquitectura Cloud de la aplicación – Servicio monitoring

```

# 2 -> Permite acceso solo a métricas de usuario
# 1 -> Permite acceso a métricas del kernel y del usuario
# 0 -> Permite acceso a datos específicos de CPU pero no trazas en bruto (Raw data)
#-1 -> Sin restricciones
tfgdenis@Master:~$ sudo sysctl -w kernel.perf_event_paranoid=2
  
```

Figura 5.7: Permisos para *perf*



Figura 5.8: Arquitectura Cloud de la aplicación – Servicio monitoring

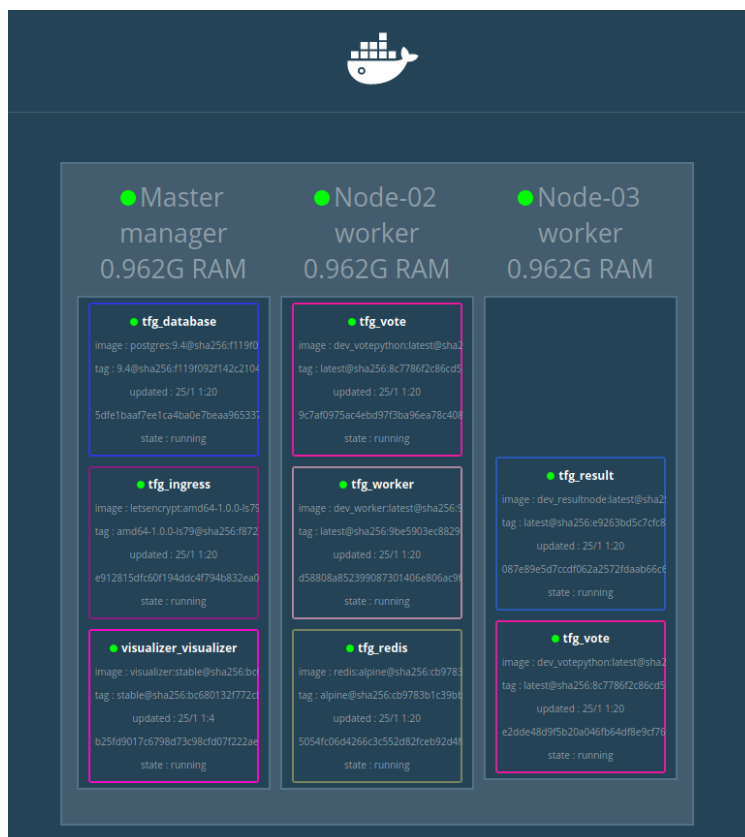


Figura 5.9: Servicio visualizer en ejecución junto al servicio tfgdennis

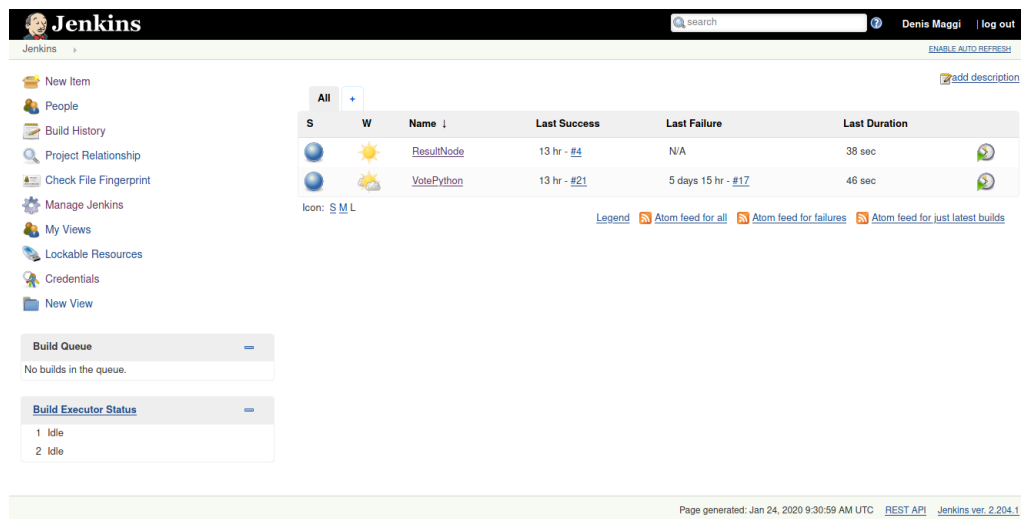


Figura 5.10: Página principal de Jenkins

5.3.4. Servicio - jenkins

Para implementar un flujo de trabajo donde las integraciones y el despliegue se realizan de manera totalmente automática necesitamos **Jenkins**. Con esta herramienta podremos crear *jobs* (tareas) que se encarguen de seguir un flujo desarrollado y finalizar dejando un registro de todo lo sucedido en el proceso. En nuestro caso hemos optado por dos tareas para implementar el *CI/CD* en nuestra arquitectura, que podemos ver en la Figura 5.10.

Dado que las tareas son similares, explicaremos una de ellas. De forma sucinta, esta tarea será la encargada de descargar el código fuente del repositorio cuando algún integrante del grupo realice una modificación y haga el cambio permanente en el repositorio del microservicio, en este caso se tratará del perteneciente a **tfgdenis** encargado de mostrar los resultados (**resultado**). Una vez descargado, creará las imágenes Docker (dos en este caso), las publicará en el repositorio de Docker Hub y actualizará el actual estado del servicio **tfgdenis** con la nueva imagen del microservicio en cuestión.

Definición de una tarea Para empezar deberemos crear una nueva tarea. Como vemos en la Figura 5.11 existen múltiples opciones para la creación de una tarea. Entraremos en detalle en **Freestyle project**.

Ejemplo ResultNode job Ahora tendremos que configurar la tarea. Estas se encuentran separadas en diferentes etapas:

- **Etap 1: General** Necesitaremos definir unas variables iniciales, que serán la *IP* de nuestro servidor **IP_MASTER** donde se encuentra ejecutando el servicio, junto con dos variables **TAG0** y **TAG1** que usaremos para agregar un *tag* a las imágenes Docker construidas.

Jenkins [Denis Maggi](#) | [log out](#)

Jenkins > All >

Enter an item name

= Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

GitHub Organization
Scans a GitHub organization (or user account) for all repositories matching some defined markers.

Multibranch Pipeline
Creates a set of Pipeline projects according to detected branches in one SCM repository.

If you want to create a new item from other existing, you can use this option:

Copy from

Page generated: Jan 24, 2020 5:43:20 PM UTC [REST API](#) [Jenkins ver. 2.204.1](#)

Figura 5.11: Opciones para la creación de tareas

String Parameter X ?

Name

Default Value

Date Parameter X ?

Name

Date Format

Default Value

Description

String Parameter X ?

Name

Default Value

Description

[\[Plain text\]](#) [Preview](#)

Figura 5.12: Definición de variables globales para la tarea

Figura 5.13: Localización del repositorio a clonar

Figura 5.14: Configuración disparador de tarea

- ***Etapa 2: Source Code Managment*** En esta etapa elegimos de donde descargaremos el código fuente necesario para la compilación de las imágenes. Como podemos ver en la Figura 5.13 necesitamos la dirección de nuestro repositorio en GitLab y las credenciales del mismo pues son repositorios privados. Para conocer más sobre cómo configurar estas credenciales en Jenkins, véase la Subsubsección B.3.2.1.
- ***Etapa 3: Build Triggers*** Debemos configurar el disparador que hará que la tarea se inicie cuando se produzca un cambio en nuestro repositorio especificado como se muestra en la Figura 5.14. Es importante tener en cuenta la *URL*, pues la usaremos para configurar nuestro GitLab como podemos ver en la Sección C.1. Además agregaremos un *token* de seguridad que lo encontramos en el botón ***Advanced***
- ***Etapa 4: Build Environment*** En este apartado marcaremos la opción

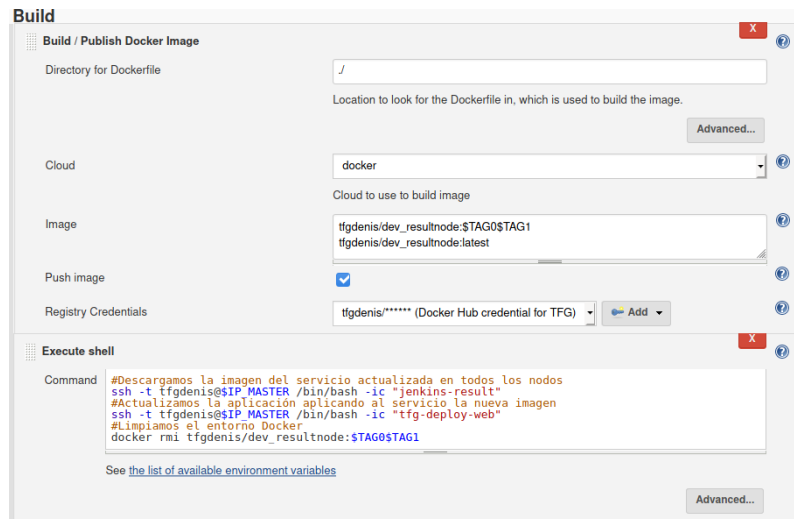


Figura 5.15: Creación de imagen, subida al repositorio Docker Hub y actualización del servicio

de borrar el entorno de trabajo antes de comenzar la tarea, pues así nos aseguramos que se hace una copia limpia e íntegra del repositorio.

- **Etap 5: Build** En la última etapa procederemos a construir la imagen, publicarla en el repositorio de Docker Hub utilizando como tags las variables globales definidas, junto a la prácticamente obligatoria *latest*. Por último en una terminal usaremos los *alias* que tenemos en el servidor principal para descargar la imagen nueva en los nodos y actualizar el servicio (Figura 5.14).

Historial y tarea en cola Pinchando en una de las tareas que vemos en la Figura 5.10 accederemos al panel resumen de dicha tarea, como vemos en la Figura 5.16 donde además de poder personalizarla, veremos el historial de trabajos ejecutados. Es muy importante revisar este panel para el caso en el que hubiera un de fallo o como estadística para el de compilación por si necesitamos alojar este servicio en un nodo con más recursos (escalado vertical).

Información tarea finalizada El último panel significativo relacionado con las tareas. Aquí encontraremos información muy importante como la revisión de lo que ha sucedido a lo largo de la tarea (*Console Output*), los parámetros de entrada usados, el último *commit* del repositorio clonado, los *tags* de las imágenes Docker subidas al repositorio Docker Hub así como la fecha y hora de la ejecución (Figura 5.17)

En el apéndice B.3 se ofrece más información sobre cómo se ha configurado *jenkins* para este trabajo.

Jenkins search Denis Maggi | log out

Jenkins > ResultNode > [ENABLE AUTO REFRESH](#)

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build with Parameters](#)

[Delete Project](#)

[Configure](#)

[Rename](#)

Project ResultNode

[add description](#)

[Disable Project](#)

[Workspace](#)

[Recent Changes](#)

Permalinks

- [Last build \(#4\), 13 hr ago](#)
- [Last stable build \(#4\), 13 hr ago](#)
- [Last successful build \(#4\), 13 hr ago](#)
- [Last completed build \(#4\), 13 hr ago](#)

Build History [trend](#)

find x

Build	Time	Started by
#4	Jan 23, 2020 8:21 PM	GitLab push by Denis Maggi
#3	Jan 19, 2020 11:51 AM	GitLab push by Denis Maggi
#2	Jan 18, 2020 6:19 PM	GitLab push by Denis Maggi
#1	Jan 18, 2020 6:18 PM	GitLab push by Denis Maggi

[Atom feed for all](#) [Atom feed for failures](#)

Page generated: Jan 24, 2020 9:32:13 AM UTC [REST API](#) [Jenkins ver. 2.204.1](#)

Figura 5.16: Panel principal de la tarea ResultNode

Jenkins search Denis Maggi | log out

Jenkins > ResultNode > #4 [ENABLE AUTO REFRESH](#)

[Back to Project](#)

[Status](#)

[Changes](#)

[Console Output](#)

[Edit Build Information](#)

[Delete build '#4'](#)

[Polling Log](#)

[Parameters](#)

[Git Build Data](#)

[No Tags](#)

[Docker Image Build / Publish](#)

[Previous Build](#)

Build #4 (Jan 23, 2020 8:21:26 PM)

Started 1 day 1 hr ago
Took [38 sec](#)

[edit description](#)

[Changes](#)

1. Fixed network ([details](#) / [gitlab](#))

Started by GitLab push by Denis Maggi

Revision: 0a4ffaef187ed98e605258bfe1ca9fa24ead708
• origin/master

Docker Image Build / Publish

Host: unix:///var/run/docker.sock
Original Container Id: 2f74e02aad3f
Committed Container Id(s): [tfgdennis/dev_resultnode:20200123_jenkins, tfgdennis/dev_resultnode:latest]

Page generated: Jan 24, 2020 9:27:15 PM UTC [REST API](#) [Jenkins ver. 2.204.1](#)

Figura 5.17: Panel resumen con el estado final de la tarea

5.4. Despliegue de contenedores Docker

Como se ha mencionado, el servidor de nuestro ejemplo de trabajo es en realidad un *cluster* formado por tres nodos (consultar cómo hacerlos en Sección C.3). Todos ellos deberán tener las herramientas necesarias instaladas.

Todos los servicios son accesibles mediante una URL segura con un certificado SSL provisto por *Let's Encrypt* (que es nuestro microservicio **ingress**). Hablaremos más en detalle de cuales son estas URLs en la Sección 5.6

Una vez definidos los servicios que pondremos en marcha en nuestro *cluster*, así como la definición de las imágenes que usaremos para crear los contenedores, es hora de definir cómo desplegaremos los servicios. Como vimos en la Sección 4.2 los contenedores se lanzan con ficheros **docker-compose.yml** que pueden contener uno o múltiples servicios. En un *cluster*, un servicio se debe lanzar con un único fichero **docker-compose.yml** en el cual se encontrarán definidos todos los microservicios que lo componen. Pasamos entonces a ver en detalle cómo desplegamos cada uno de los servicios implementados.

Nota: Para conocer el comando con el cual lanzamos los servicio, véase la Subsubsección B.1.0.2.

5.4.1. Servicio - tfgdennis

Nuestra aplicación principal despliega cada uno de sus microservicios mediante contenedores. Debido a la posibilidad de desplegar los microservicios en diferentes nodos del *cluster*, tendremos algunas peculiaridades a la hora de definir el lanzamiento de nuestro servicio; algunos de estos microservicios sólo podrán ser lanzados en el mismo nodo e incluso otros los podremos tener replicados varias veces. Pasamos a explicar el porqué y cuáles son los microservicios con éstas características.

ingress Se trata de la entrada a la aplicación. Se ha decidido considerar la importancia de la seguridad, es por ello que hemos registrado un dominio. Este dominio se encuentra asociado únicamente a una IP **FIJA** que será la del nodo *manager* (explicaremos que es en la Sección 5.5) y por ésta configuración nos vemos obligados a lanzarlo en dicho nodo. Existe una solución a este problemas mediante redirección de IP.

db Dado que la base de datos se guarda en un volumen creado por Docker (Subsección A.1.3) no tenemos forma de compartir el contenido de éstos entre las diferentes instancias de microservicio desplegadas en los diferentes nodos. Se puede solucionar mediante la implementación de un sistema de ficheros en red NFS (*Network File System*), compartiendo el contenido de los volúmenes por los diferentes nodos.

votos En este caso, podremos replicar el servicio tantas veces como queramos, para poder tener un mejor abastecimiento de la aplicación de cara a los clientes. Dado que en algún momento, podríamos vernos desbordados por una alta participación de personas en el mismo instante, es recomendable repartir la carga que puede producir procesar miles de peticiones en pocos segundos entre varios nodos.

Ahora describiremos algunas líneas del archivo `docker-compose.yml`, ubicado en `DevOps/ServerConf/`, necesario para desplegar como servicio nuestra aplicación (ver la Figura 5.18). Veremos las reglas para forzar el lanzamiento de un servicio en un nodo u otro, según su nombre o su *rol*. También impondremos una política de actualización de réplicas en caso de actualizar las imágenes de los contenedores, para que éstas lo hagan de manera progresiva y así evitar quedarnos sin servicio. Por último veremos las políticas en caso de fallo, cómo y cuando reiniciarse y otras curiosidades.

*Nota: Se anima al lector a consultar los ficheros originales que se adjuntan al trabajo. Hemos omitido la configuración de volúmenes en **ingress** pues la detallaremos mejor en Sección 5.7 y muchas otras configuraciones que tienen menor importancia.*

Para poder lanzar el servicio ejecutaremos en el nodo *Manager* el siguiente comando. El servicio lo llamaremos **tfg** para abreviar y poder verlo mejor.

```
tfgdenis@Master:~$ docker stack deploy -c docker-compose.yml tfg
```

A continuación podemos ver un ejemplo de los contenedores que se encuentran desplegados en el nodo *Manager* tras lanzar el servicio **tfgdenis** mediante el comando `docker ps`. Únicamente veremos dos contenedores, pues los demás están desplegados en los demás nodos. Notaremos que los nombres de los contenedores son distintos a los definidos en el `docker-compose.yml`

```
tfgdenis@Master:~$ docker ps
IMAGE                PORTS                NAMES
postgres:9.4         5432/tcp             tfg_database.1.w7ib66vtbkk6p187b2486xpmc
linuxserver/letsencr... 80/tcp, 443/tcp     tfg_ingress.y8ytt6w4am7qqf6v4afjq908n.bdlq4aqs...
```

El comando `docker stack ps tfg` nos muestra el estado del servicio en los diferentes nodos, así podremos conocer dónde se encuentra cada microservicio sin tener que visitar cada nodo. Podemos ver un ejemplo en la Figura 5.19

5.4.2. Servicio - monitoring

Este servicio se deberá lanzar siempre de la misma manera, es decir, definiremos para cada microservicio en qué nodo debe alojarse, pero lo haremos por el nombre

```

version: '3.8'
services:
  ingress:
    image: linuxserver/letsencrypt:amd64-1.0.0-ls79
    environment: #Variables de entorno para Let's Encrypt y Cerbot
    #(...)
    volumes: #Configuraciones propias para Nginx.
    #(...)
    ports: #Puertos expuestos al exterior del Cluster
    - 443:443
    - 80:80
    deploy: #Políticas de despliegue
    mode: global
    restart_policy: #Políticas de reinicio
    condition: any #En caso de cualquier fallo/reinicio
    delay: 10s
    placement: #Desplegar solo en el nodo líder
    constraints: [node.role == manager]
    networks: #Redes a las que se conecta el servicio
    - tfg
    - frontend
    - backend
  redis:
    image: redis:alpine
    #(...)
  database:
    image: postgres:9.4
    #(...)
    deploy:
      placement:
        constraints: [node.role == manager]
  vote:
    image: tfgdenis/dev_votepython:latest
    deploy:
      replicas: 2
      placement:
        max_replicas_per_node: 1
    #(...)
  result:
    image: tfgdenis/dev_resultnode:latest
    #(...)
  worker:
    image: tfgdenis/dev_worker:latest
    deploy:
      placement:
        constraints: [node.role != manager]
  networks:
    #(...)
    tfg:
      external: true
      name: tfg
    #(...)

```

Figura 5.18: Fichero docker-compose.yml para desplegar el servicio tfgdenis

```
tfgdenis@Master:~$ docker stack ps tfg
```

NAME	IMAGE	NODE
tfg_ingress.y8ytt6w4am7qqf6v4afjq908n	linuxserver/letsencrypt:amd64-1.0.0-ls79	Master
tfg_database.1	postgres:9.4	Master
tfg_redis.1	redis:alpine	Node-02
tfg_worker.1	tfgdenis/dev_worker:latest	Node-02
tfg_result.1	tfgdenis/dev_resultnode:latest	Node-03
tfg_vote.1	tfgdenis/dev_votepython:latest	Node-03
tfg_vote.2	tfgdenis/dev_votepython:latest	Node-02

Figura 5.19: Ejemplo ejecución `docker stack ps tfg`

del nodo utilizando como ya hemos visto la opción `[node.hostname ==]` y no por su *rol*, así nos aseguramos de que siempre se despliega de la misma manera. Esto se debe a que cada nodo deberá tener ejecutándose como mínimo los microservicios de `node-exporter` y `cadvisor` cada uno, pues son los que capturan las métricas de los nodos, tal y como vimos en la Subsección 5.3.2. En cambio, los microservicios de `grafana` y `prometheus` deberán desplegarse siempre sobre el nodo con el *rol* de *Manager*. En la Figura 5.20 vemos una versión simplificada del archivo `DevOps/ServerConf/Monitoring/docker-compose.yml` donde se encuentran definidos cómo deben ser lanzados los microservicios. Es importante la configuración añadida en los volúmenes de `prometheus` en la Sección 5.6.

5.4.3. Servicio - jenkins

Este servicio es sencillo de desplegar, pues no requiere de ninguna configuración añadida. Solo hay que tener en cuenta que, como decíamos en la Sección 5.2 usaremos la herramienta `docker` dentro del contenedor, por tanto, para que *Jenkins* tenga acceso a las mismas imágenes que el nodo *Manager* donde se ejecutará Jenkins, tendremos que mapear el socket de Docker. También crearemos un volumen donde se guardarán todas las configuraciones y tareas que ejecutemos en el servicio. A continuación vemos el archivo `DevOps/ServerConf/Jenkins/docker-compose.yml`

```
version: '3.4'
services:
  jenkins:
    image: tfgdenis/jenkins:latest
    volumes: #Volumen personal y mapeo de docker.sock
      - jenkins_data:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
    networks: [frontend]
    deploy:
      placement:
        constraints: [node.role == manager]
volumes:
  jenkins_data:
networks:
  frontend:
    external:
      name: tfg
```



```

version: '3.2'
services:
  grafana:
    image: grafana/grafana:latest
    volumes: [grafana-data:/var/lib/grafana]
    deploy:
      placement:
        constraints: [node.role == manager]
      networks: [backend]
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus:/prometheus
    deploy:
      placement:
        constraints: [node.role == manager]
      networks: [backend]

# NODE-EXPORTER # IMAGE: prom/node-exporter:latest
manager-01:
  volumes: [/proc:/host/proc:ro, /sys:/host/sys:ro, /:/rootfs:ro]
  command: #(...)
  deploy:
    restart_policy:
      condition: on-failure
    placement:
      constraints:
        - node.role == manager
    networks: [backend]
node-02: #(...)
node-03: #(...)

# CADVISOR # IMAGE: google/cadvisor:latest
docker-manager-01:
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker:/var/lib/docker:ro
  deploy:
    restart_policy:
      condition: on-failure
    placement:
      constraints: [node.role == manager]
    networks: [backend]
docker-node-02: #(...)
docker-node-03: #(...)

volumes:
  grafana-data:
  prometheus:

networks:
  backend:

```

Figura 5.20: Configuración docker-compose.yml del servicio *monitoring*

5.4.4. Servicio - visualizer

Este servicio se debe lanzar en el nodo *manager* al igual que ocurría con *Jenkins*, dado que nos dará información sobre los contenedores y requiere del acceso al *socket* de Docker. A continuación vemos el archivo encargado de definir las instrucciones para su despliegue: DevOps/ServerConf/Visualizer/docker-compose.yml:

```
version: "3"
services:
  visualizer:
    image: dockersamples/visualizer:stable
    ports: [8080:8080]
    volumes: [/var/run/docker.sock:/var/run/docker.sock]
    deploy:
      placement:
        constraints: [node.role == manager]
```

5.5. Orquestación - Docker Swarm

Una vez tengamos *DigitalOcean* creado con tres nodos, pasaremos a configurar *Docker Swarm* que será el encargado orquestar todos nuestros servicios, es decir, todos los contenedores que despliegamos. *Swarm* utiliza una jerarquía sencilla, donde existen dos tipos de nodos:

Manager Es el encargado de gestionar los nodos esclavos/trabajadores o *workers* mediante el envío de *tasks*.³. También es el encargado de realizar las tareas necesarias para mantener el estado del *cluster* deseado.

Worker Recibe y ejecuta tareas enviadas por los *Managers*. Por defecto todos los nodos son *Workers* incluidos los nodos asignados como *Managers*, pero podemos modificar esta configuración si así lo queremos.

Para poder indicar qué *rol* adquiere cada nodo deberemos configurarlos a través de la herramienta **docker** desde la terminal.

Primero iniciaremos el nodo *Manager* que será el encargado de gestionar los demás; para ello desde nuestro nodo **tfgdenis@master** iniciaremos el servicio *Swarm* de la siguiente manera:

```
docker swarm join --advertise-addr $IP_MANAGER
```

³Tareas creadas por el nodo *Manager* y enviadas a los *Workers*

El puerto por defecto usado por *Swarm* para conectar los nodos es el 2377. Así pues tras configurar el nodo actual como *Manager* Docker nos mostrará el comando necesario para ejecutarlo en los demás nodos y agregarlos al *Swarm* como *Worker*. A continuación explicaremos cómo obtener el comando que habría que ejecutar para añadir un nodo con alguno de los dos roles descritos anteriormente.

```
#Nos muestra el comando necesario para añadir a nuestro Swarm un nodo con rol Worker
docker swarm join-token worker
#Ejemplo de salida (token simplificado) (IP nodo Manager simplificada)
docker swarm join --token SWMTKN-1-2qef53s45-bxlegz9ofin IP_MASTER:2377
```

```
#Nos muestra el comando necesario para añadir a nuestro Swarm un nodo con rol Manager
docker swarm join-token manager
#Ejemplo de salida (token simplificado) (IP nodo Manager simplificada)
docker swarm join --token SWMTKN-1-2qef53s45-bxlegzoasij IP_MASTER:2377
```

5.6. Scripts y Configuraciones

En esta sección veremos un script, ubicado en `DevOps/Scripts/Server` pensado para facilitar la tarea del perfil *DevOps* en caso de que desee descargar o actualizar las imágenes de sus servicios de forma manual en todos sus nodos. También veremos las configuraciones que hemos realizado a los diferentes servicios mencionados en la Sección 5.4.

update-images.sh (Figura 5.21) Con este script podremos descargar o actualizar todas la imágenes que componen nuestra aplicación (la versión `latest`), así como la última versión de todas las imágenes del servicio `monitoring`. En este *script* tenemos en cuenta que los microservicios `prometheus` y `grafana` pertenecientes al servicio `monitoring` no se deben descargar en los nodos *Worker* pues nunca se desplegarán dichas imágenes en esos contenedores.

5.6.1. Configuración de dominios

Hemos creado un dominio para servir la aplicación y sus diferentes servicios. Vamos a describir brevemente cuales son los dominios que disponemos y que se sirve en cada uno.

- **tfg.eus** En esta URL encontraremos la aplicación principal, servimos directamente el microservicio `votos` del servicio principal `tfgdenis`.
- **tfg.eus/result** Encontramos los resultados de nuestra aplicación tras votar. Servimos microservicio `resultados` del servicio principal `tfgdenis`.

```
#!/bin/bash
#Comprobamos que exista argumento
if [ ! $1 ] ; then
    ...
    exit 2
fi
#Descargamos imagen:latest en todos los nodos
function pull-image(){
    docker pull ${image}:latest
    parallel-ssh -t 0 -i -h $file -x '-i ~/.ssh/id_generic' "docker pull ${image}:latest"
}
#Descargamos una imagen con un tag especificado
function pull-image-tag(){
    docker pull ${image}:${1}
    parallel-ssh -t 0 -i -h $file -x '-i ~/.ssh/id_generic' "docker pull ${image}:${1}"
}
#Limpiamos el entorno Docker de todos los nodos
function prune-all(){
    docker system prune -f
    parallel-ssh -t 0 -i -h $file -x '-i ~/.ssh/id_generic' "docker system prune -f"
}
#Ejemplo en caso de elegir opción "resultnode" o "monitoring"
case $1 in
    resultnode)
        file="${DIR_HOSTS}/all" #Archivo con las IPs de los nodos del cluster
        image="tfgdenis/dev_resultnode" #Imagen a descargar/actualizar
        pull-image &
        if [ $2 ] ; then
            pull-image-tag $2
        fi
        prune-all
    ;;
    votepython) ...
    worker) ...
    monitoring)
        file="${DIR_HOSTS}/all"
        declare -a listImages=("grafana/grafana" "prom/prometheus" "prom/node-exporter"
        ↪ "google/cadvisor")
        length=${#listImages[@]}
        for (( i=0; i<${length}; i++ )); #i in "${listImages[@]}"
        do
            image=${listImages[$i]}
            echo -e "\e[92;1m[*] Image: $image \e[0m"
            if [ $i -gt 1 ]
            then
                pull-image
            else
                docker pull ${image}:latest
            fi
        done
        prune-all
    ;;
    *) ...
esac
```

Figura 5.21: Script update-image.sh para el servidor.

- `grafana.tfg.eus` En este subdominio encontraremos el microservicio de grafana, perteneciente al servicio de monitoring.
- `jenkins.tfg.eus` En este subdominio encontramos el servicio de jenkins

5.6.2. Configuración ingress - tfgdenis

Este microservicio es en realidad el más importante para poder servir la aplicación y todos los servicios que disponemos, pues es el encargado de redirigir todas las peticiones exteriores que llegan al *cluster* a los diferentes servicios que dispongamos. En este caso lo desplegamos junto a nuestra aplicación principal, pero podríamos separarlo en caso de que fuera necesario. Ya hemos visto en la Subsección 5.4.1 el despliegue general del servicio, pero pasaremos a explicar un poco más en detalle las configuraciones de `environment` y las distintas configuraciones que agregamos al contenedor mediante la opción de `volumenes`. Para entender mejor como funciona el mapeo de volúmenes en Docker consúltase la Subsección A.1.3.

```
environment:  #Variables de entorno para Let's Encrypt y Cerbot
- URL=tfq.eus  #Nuestra URL
- SUBDOMAINS=www,grafana,jenkins  #Subdominios
- VALIDATION=dns  #Tipo de validación
- DNSPLUGIN=digitalocean  #Acceso a la API
- DHLEVEL=2048
- STAGING=false  #Generación de certificados para desarrollo
volumes:  #Configuraciones propias para Nginx.
- ./config:/config  #Configuración general
- ./config_docker/digitalocean.ini:/config/dns-conf/digitalocean.ini  #Clave API DigitalOcean
- ./config_docker/nginx.conf:/config/nginx/site-confs/default  #Configuración principal
- ./config_docker/subdomains/grafana.subdomain.conf:/config/nginx/
↳ proxy-confs/grafana.subdomain.conf  #Configuración para subdominio Grafana
- ./config_docker/subdomains/jenkins.subdomain.conf:/config/nginx/
↳ proxy-confs/jenkins.subdomain.conf  #Configuración para subdominio Jenkins
```

La siguiente configuración pertenece a las variables de entorno del sistema especificadas mediante la opción `environment` en el fichero `DevOps/docker-compose.yml`:

- `URL=tfq.eus` Dominio principal.
- `SUBDOMAINS=www,grafana,jenkins` Subdominios configurados para acceder a los diferentes servicios.
- `VALIDATION=dns` → Validación para *Let's Encrypt*.
- `DNSPLUGIN=digitalocean` Obligatorio si configuramos la validación como `dns`.
- `STAGING=false` En caso de iniciar el servicio en desarrollo donde probaremos múltiples configuraciones hasta dar con la apropiada, deberemos utilizar esta opción y configurarla a `true`, pues *Let's Encrypt* nos dará certificados en modo *staging*.

Las siguientes configuraciones pertenecen a las configuraciones impuestas a *Nginx* mediante el mapeo de volúmenes con `volumes` correspondientes al fichero ubicado en `DevOps/docker-compose.yml`. Todas las configuraciones se encuentran ubicadas en el directorio `DevOps/ServerConf/config_docker/`

nginx.conf Es la configuración más importante del servicio, es donde definiremos todas las reglas para redirigir cada petición al microservicio que le corresponde.

```
#Lista de los "servidores" disponibles
upstream app_servers_vote {
    server tfg_vote:5000;
}
upstream app_servers_result {
    server tfg_result:5001;
}
upstream app_servers_grafana {
    server monitoring_grafana:3000;
}
upstream app_servers_jenkins {
    server jenkins_jenkins:8080;
}

# Todo el tráfico HTTP lo redirigimos a HTTPS
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name tfg.eus;
    return 301 https://$host$request_uri;
}

#Bloquee de servidor principal
server {
    #Escuchamos puerto 443
    listen 443 ssl; #http2 default_server;
    root /config/www;
    index index.html index.htm index.php;

    #Agregamos la configuración SSL
    include /config/nginx/ssl.conf;
    #Redireccionamiento al servicio de votos en tfg.eus
    location / {
        proxy_pass      http://app_servers_vote;
        proxy_redirect  off;
        # ...
    }
    #Redireccionamiento al servicio de resultados en tfg.eus/result
    location /result {
        proxy_pass      http://app_servers_result;
        proxy_redirect  off;
        # ...
    }
    #Necesario para indicar dónde se encuentra el código fuente dentro del árbol de
    ↪ directorio
    location /voteSources {
        proxy_pass      http://app_servers_result;
        proxy_redirect  off;
        # ...
    }
}

#Incluimos las configuraciones de los subdominios (grafana y jenkins)
include /config/nginx/proxy-confs/*.subdomain.conf;
```

Podemos destacar de esta configuración la lista de los servidores disponibles al principio del fichero. En una configuración normal sin contenedores dispo-

nibles, cada uno de los `upstream` se refiere a un microservicio y dentro de estos colocaríamos todos los nodos físicos que sirven dicho microservicio. Es así como *Nginx* puede ser utilizado como balanceador de carga, pues en función de la cantidad de peticiones que lleguen para un mismo `upstream` éste las distribuye entre los diferentes nodos, repartiendo la carga. Dado que nosotros utilizamos *Docker Swarm* esto no será necesario, pues integra una función de balanceo entre las diferentes réplicas que lancemos. Veremos un poco más en detalle cómo puede hacer esto en la Sección 5.7.

subdomains/grafana.subdomain.conf Para el subdominio es importante que el microservicio `grafana` no esté usando ninguna URL base en la configuración y que se encuentren correctamente configurados los subdominios. Ver la Sección C.3 para conocer la configuración usada.

```
server {
    #Acceso por el puerto seguro HTTPS
    listen 443 ssl;
    #Importante configurar correctamente el subdominio al que pertenece grafana.tfg.eus
    server_name grafana.*;
    include /config/nginx/ssl.conf;
    location / {
        include /config/nginx/proxy.conf;
        #Nombre del upstream en la configuración principal nginx.conf
        proxy_pass      http://app_servers_grafana;
        proxy_redirect  off;
    }
}
```

subdomains/jenkins.subdomain.conf Para este subdominio debemos tener correctamente configurada la URL de acceso en la configuración del servicio `jenkins`.

```
server {
    #Acceso por el puerto seguro HTTPS
    listen 443 ssl;
    #Importante configurar correctamente el subdominio al que pertenece jenkins.tfg.eus
    server_name jenkins.*;
    include /config/nginx/ssl.conf;
    location / {
        include /config/nginx/proxy.conf;
        #Nombre del upstream en la configuración principal nginx.conf
        proxy_pass      http://app_servers_jenkins;
        proxy_redirect  off;
    }
}
```

5.6.3. Configuración Prometheus - monitoring

Como vimos en la sección anterior, la principal configuración proviene del archivo `nginx.conf` que es el encargado de realizar la tarea de recibir las peticiones del exterior y redireccionarlas correctamente a cada microservicio. Pero en este caso, el microservicio `prometheus` tendrá una configuración propia extra para poder conocer

Prometheus Alerts Graph Status Help					
Targets					
All Unhealthy					
cadvisor (3/3 up) show logs					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://docker-manager-01:8080/metrics	UP	instance="docker-manager-01:8080" job="cadvisor"	7.869s ago	113.1ms	
http://docker-node-02:8080/metrics	UP	instance="docker-node-02:8080" job="cadvisor"	9.453s ago	103.4ms	
http://docker-node-03:8080/metrics	UP	instance="docker-node-03:8080" job="cadvisor"	11.663s ago	128.9ms	
node-exporter (3/3 up) show logs					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://manager-01:9100/metrics	UP	instance="manager-01:9100" job="node-exporter"	1.974s ago	47.18ms	
http://node-02:9100/metrics	UP	instance="node-02:9100" job="node-exporter"	1.451s ago	41.52ms	
http://node-03:9100/metrics	UP	instance="node-03:9100" job="node-exporter"	4.39s ago	35.93ms	

Figura 5.22: Objetivos de Prometheus de los cuales adquiere las métricas.

cómo se llaman los microservicios de los que tiene que extraer las métricas que nos interesan; este fichero lo encontramos en DevOps/ServerConf/Monitoring/ y se trata del fichero `prometheus.yml` el cual mapeamos en la opción `volumes` como vimos en la Subsección 5.4.2. Pasaremos a ver la configuración un poco más en detalle.

```
#Configuración global
global:
  scrape_interval:      15s #Configuramos cada cuánto extraemos información
  evaluation_interval: 15s #Tiempo de evaluación de reglas
  #Se utiliza si configuramos un servicio de alertas.
  external_labels:
    monitor: 'codelab-monitor'
#Aquí definimos los endpoints de los cuales extraemos la información que nos hace falta
#para monitorizar.
scrape_configs:
  - job_name: 'node-exporter'
    static_configs:
      # the targets listed here must match the service names from the docker-compose file
      - targets: ['manager-01:9100', 'node-02:9100', 'node-03:9100']
  - job_name: 'cadvisor'
    static_configs:
      # the targets listed here must match the service names from the docker-compose file
      - targets: ['docker-manager-01:8080', 'docker-node-02:8080', 'docker-node-03:8080']
```

La parte más importante en este fichero de configuración se halla en la definición los destinos finales a los cuales `prometheus` debe conectarse para extraer las métricas y esto lo hacemos en `scrape_configs`. En este caso definimos los dos microservicios de los que disponemos y lo más importante indicamos el nombre de estos microservicios que definimos en la Subsección 5.4.2. Como muestra de las comunicaciones con los microservicios, hemos mapeado por un momento el puerto en el servicio para poder acceder a él desde fuera y hacer una captura (ver Figura 5.22).

5.7. Infraestructura de la red

En esta sección detallaremos la infraestructura de red que hace posible la comunicación entre los diferentes servicios e internamente entre los microservicios. Lo veremos

de manera más clara con un pequeño gráfico donde visualizaremos cada uno de los microservicios que componen el servicio y veremos cómo ocurre exactamente el balanceo de carga en los servicios que disponen de réplicas en sus microservicios.

Antes de nada tendremos que tener en cuenta la forma en la que *Docker Swarm* se estructura cuando lanza un servicio. Al crear un **stack** (es la manera de lanzar un servicio usando los **docker-compose.yml** vistos anteriormente) le asignamos un nombre; este será el nombre del servicio. *Docker Swarm* añade a cada contenedor lanzado del *stack* (que viene especificado en el **docker-compose.yml**) su nombre (el nombre de ese microservicio). Dado que estos microservicios pueden ser replicados, deben diferenciarse de alguna manera y es aquí donde *Docker Swarm* agrega un *hash*⁴ al nombre del contenedor.

La cuestión es cómo referirnos a ese contenedor, si no sabemos cómo se llamará más allá de la estructura inicial descrita como **nombreStack_nombreMicroservicio.hash**, pues bien, internamente *Docker Swarm* usará una tabla de enrutamiento y un servicio *DNS* propio, que nos permite referirnos al contenedor como su estructura inicial (**nombreStack_nombreMicroservicio**) y él internamente enruta al contenedor que crea conveniente. Es aquí donde aprovechamos la capacidad de *Docker Swarm* para ofrecernos un balanceador de carga totalmente automatizado sin requerir de ninguna configuración extra.

5.7.1. Servicio tfgdenis

Como podemos ver en la Figura 5.23 este servicio posee todos sus microservicios totalmente blindados al exterior, de esta forma sólo exponemos dos puertos bien conocidos y controlamos todo el tráfico entrante y saliente. Además vemos cómo podemos acceder a los dos microservicios que nos muestran los resultados de los votos y la web para votar desde la misma URL.

5.7.2. Servicio monitoring

Del mismo modo que ocurre con el servicio anterior, en **monitoring** todos los microservicios se encuentran blindados y el único que se comunica con el **ingress** será **grafana**, el visualizador de métricas. Como vemos en la Figura 5.24 el acceso a éste se realiza mediante un subdominio.

5.7.3. Servicio jenkins

Por último tenemos el servicio de *Jenkins*, que en realidad no tiene ninguna complejidad, pues se trata de un único servicio bastante sencillo de desplegar, debido a

⁴Función computable utilizada para transformar un bloque de datos en otro bloque *aleatorio*

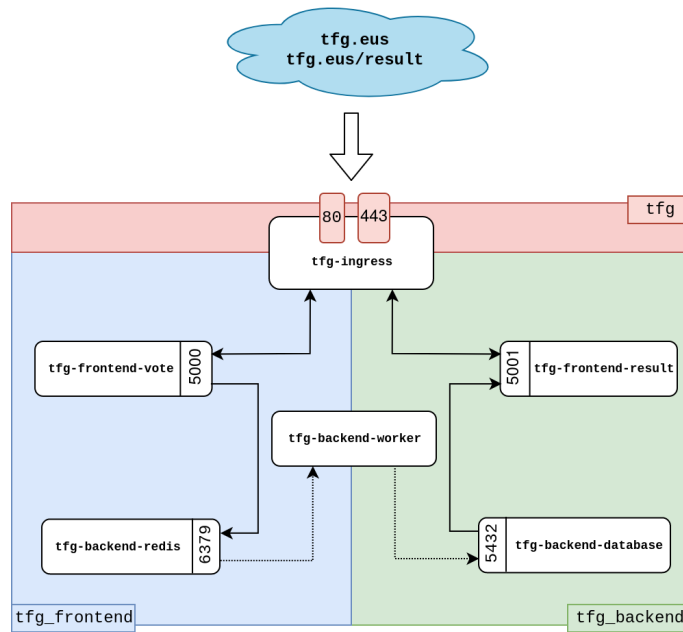


Figura 5.23: Infraestructura de red de la aplicación – Servicio **tfgdenis**

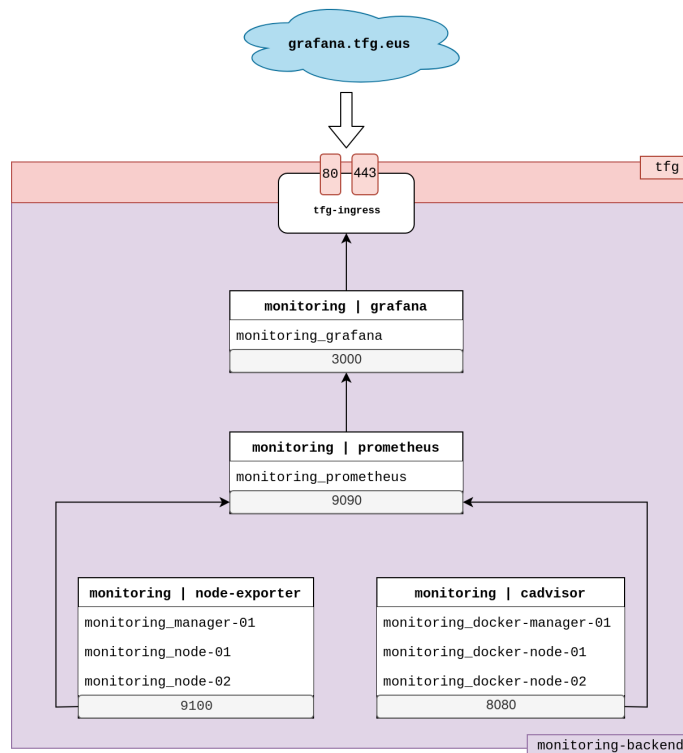


Figura 5.24: Infraestructura de red del servicio **monitoring**

no depender de otros microservicios como vemos en la Figura 5.25. El acceso a éste se realiza mediante un subdominio.

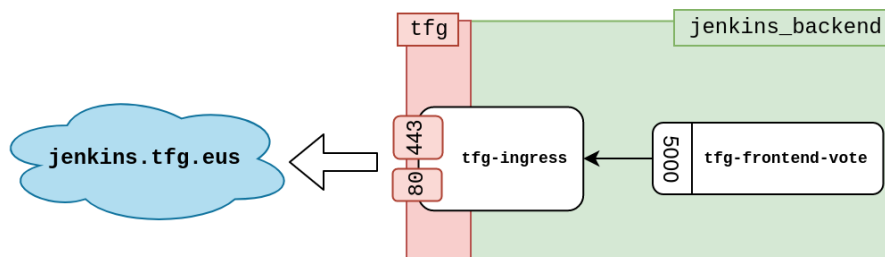


Figura 5.25: Infraestructura de red del servicio `jenkins`

PUERTO	PROTOCOLO	DESCRIPCIÓN
2376	tcp	Comunicación segura con el cliente Docker. Necesario para utilizar Docker Machine
2377	tcp	Comunicación entre los nodos de Docker Swarm. Solo abrir en Managers.
4789	udp	Comunicación entre nodos (descubrimiento por red de contenedores). Necesario abrirlo en todos los nodos
7946	tcp/udp	Capa de tráfico de red. Necesario abrirlo en todos los nodos

Tabla 5.1: Puertos abiertos para el servicio Docker Swarm

5.8. Seguridad

Un aspecto muy importante cuando nos referimos a aplicaciones *Cloud* es la seguridad, pues se trata de exponer una aplicación completa a través de internet, donde millones de personas tendrán acceso a ellas con un simple click. Es por ello, que una de las medidas más importantes del proyecto es la propia contenerización de cada uno de los servicios, para que, en caso de ataque a uno de éstos, cada uno de ellos se encuentre aislado y el daño sea el menor posible.

Además de las precauciones tomadas, descritas en la Sección 5.7 respecto a la red y la exposición de los puertos imprescindibles, es cierto que para el uso de *Docker Swarm* se utilizan puertos específicos para que los nodos puedan comunicarse y realizar los despliegues oportunos. En la Tabla 5.1 se resume, tal y como se indica en la documentación oficial de Docker (*Getting started with swarm mode*, 2020) los puertos expuestos y para qué son utilizados.

Por consiguiente hemos agregado unas reglas en el *Firewall*⁵ de nuestros nodos, para evitar que posibles atacantes exploren vulnerabilidades emergentes. Podemos ver un prototipo de la configuración en la Figura 5.26. Configuraremos los puertos para que únicamente las diferentes IPs que componen nuestro *Cluster* tengan acceso a dichos puertos.

Nota: ya que nos encontramos agregando reglas en el Firewall, agregaremos una para el puerto 22 que corresponde al puerto para el acceso a los nodos mediante SSH.

Además, utilizamos una imagen específica basada en *Nginx* para la certificación de nuestro dominio, mediante un certificado **SSL** que nos provee *Lets's Encrypt*. El

⁵Cortafuegos. Diseñado para bloquear acceso no autorizado.

```

#MANAGER (<manager-ip>)
ufw allow 22/tcp
ufw allow 2377/tcp
ufw allow from <worker1-ip> to any port 7946
ufw allow from <worker2-ip> to any port 7946
ufw allow from <worker1-ip> to any port 4789 proto udp
ufw allow from <worker2-ip> to any port 4789 proto udp

#NODE1 (<worker1-ip>)
ufw allow 22/tcp
ufw allow from <manager-ip> to any port 7946
ufw allow from <worker2-ip> to any port 7946
ufw allow from <manager-ip> to any port 4789 proto udp
ufw allow from <worker2-ip> to any port 4789 proto udp

#NODE2 (<worker2-ip>)
ufw allow 22/tcp
ufw allow from <manager-ip> to any port 7946
ufw allow from <worker1-ip> to any port 7946
ufw allow from <manager-ip> to any port 4789 proto udp
ufw allow from <worker1-ip> to any port 4789 proto udp

```

Figura 5.26: Esquema protección *Firewall*

servicio encargado es nuestro ya mencionado *ingress*, el cual utiliza mediante la herramienta *Cerbot* certificar nuestro dominio y mantener dichos certificados actualizados, renovándolos cada vez que éstos caducan de manera automática, es decir, cada 30 días.

5.9. Flujo de trabajo en el servidor

El flujo de trabajo del servidor es relativamente sencillo. A modo de resumen una vez tenemos las imágenes Docker de nuestros servicios en local, con sus códigos fuentes actualizados tenemos dos maneras de seguir el flujo. La más sencilla y automatizada es subir los cambios a *GitLab* y automáticamente el servicio de *jenkins* se encargará de generar las imágenes y desplegar dicho servicio en nuestro *cluster* mediante los *Dockerfiles* y los *docker-compose.yml* definidos con ese fin. La otra posibilidad es en caso de querer testear alguna funcionalidad en particular en el cluster que no estemos muy seguros de que pueda llegar a funcionar de la misma manera o simplemente porque estemos realizando cambios concretos de configuraciones de código que podrían afectar al rendimiento. En tal caso con los scripts proporcionados, podríamos subir de manera manual las imágenes a los repositorios de Docker Hub y modificar las reglas del despliegue de los servicios oportunos para lanzar dichas imágenes.

5.9.1. Alta disponibilidad y la escalabilidad

Como hemos visto, nuestra metodología simplifica enormemente el despliegue de las aplicaciones, de manera que con tan sólo unos cambios de configuración podemos

desplegar varias instancias del mismo microservicio para ampliarlo y escalarlo según las necesidades que tengamos. Esto nos permite distribuir la carga entre los diferentes nodos que formen nuestro *cluster*. Ahora bien, es muy importante conocer cuales son los requisitos de cada uno de los microservicios que tenemos, pues si bien es fácil replicarlos, no tenemos verdadero control (a menos que lo configuremos) sobre en qué nodos se despliega qué, es decir, si tenemos varios microservicios que requieren mucho cómputo cuando son invocados y los replicamos, tenemos que asegurarnos que como mínimo no se repliquen dos de ellos en el mismo nodo, pues perdería la lógica de la escalabilidad y de la alta disponibilidad en cuanto se requiriera el uso de los 2 microservicios. Por ejemplo, supongamos que tenemos un microservicio que requiere de mucho cómputo y asignamos dos replicas del mismo; han entrado dos peticiones simultaneas, *Docker Swarm* las repartirá a cada una de las réplicas, pero al encontrarse en el mismo nodo es muy probable que lo colapsemos y *se caiga*⁶. Por eso es muy importante realizar pruebas de carga además de las pruebas de tests unitarios al código de la aplicación.

5.9.2. Despliegue Continuo (CI/CD)

A lo largo de este trabajo, nos hemos inclinado por adoptar el CI/CD para poder dotar a la empresa de flexibilidad, rapidez y eficiencia. Lo hemos conseguido gracias a la ayuda de los contenedores y a *Jenkins* que se encarga de integrar y desplegar de manera continua todo el trabajo realizado en la empresa.

5.10. Mantenimiento y nuevo desarrollo

Para finalizar el capítulo dejamos unas pequeñas pautas a seguir durante el desarrollo y el mantenimiento de las aplicaciones que se van generando en la empresa. Es de vital importancia mantener nuestros servidores actualizados. Tenemos que actualizar cada uno de nuestros nodos obligatoriamente en caso de una mejora en el motor de Docker o en caso de paquetes de seguridad que corrigen vulnerabilidades.

Además no olvidemos que si bien nuestros servidores están bajo sistema *Linux*, todos los contenedores lo están también, así pues, aunque no existan cambios en nuestro código fuente, hay que revisar con regularidad las actualizaciones que sufran las imágenes base que utilizamos en ellas.

En caso de que el proyecto se vea incrementado en cuanto a servicios que ofrecer, tendremos que desarrollar nuevo código, nuevos entornos para nuestras imágenes, modificar los scripts para que contemplen el nuevo servicio, así como agregar tareas si fuera necesario implementar CI/CD con estas.

⁶Referente a cuando una máquina se queda sin memoria RAM o se encuentra muy saturada, comienzan los fallos de paginación y el computador es inestable, hasta el punto en el que puede dejar de dar servicio y reiniciarse.

Capítulo 6

Conclusiones

En este trabajo fin de grado hemos usado una pequeña parte de la gran potencia que los contenedores nos pueden ofrecer, en un contexto de producción de software, y a su vez hemos podido profundizar en aspectos importantes de configuraciones que pueden ser trasladadas a otros proyectos con diferentes servicios y aplicaciones.

Se ha detallado cómo aplicar la Integración Continua y el Despliegue Continuo (CI/CD) utilizando como recursos principales servidores con CPUs virtualizadas (vCPU) de bajo coste y posibilidades de escalado horizontal, de una forma económica, manejable y rápida.

Como líneas futuras se podría automatizar test unitarios para la aplicación. Gracias a la herramienta Jenkins podríamos ejecutar y probar el software, comunicarlo con el servidor de producción y realizar una actualización progresiva de las diferentes réplicas que tengamos en nuestro servicio. De esta forma, el cliente nunca vería una caída significativa del servicio y disfrutaría de las últimas novedades.

Anexos

Anexo A

Instalación de Herramientas

A.1. *docker* y *docker-compose*

Docker se encuentra disponible tanto para plataforma *Linux*, *MacOS* como *Windows* pero en nuestro caso usaremos *Linux* como sistema operativo. Tal y como se ha mencionado anteriormente, esta instalación sólo será válida para una plataforma con Sistema Operativo Linux. Para la instalación local usamos una distribución Linux basada en Arch, *Manjaro 18.1.5 Juh Raya* y para la plataforma Cloud usaremos Linux basado en Debian, *Ubuntu 18.04.3 LTS*. Además instalaremos la version ***Docker-CE*** (*Docker Engine - Community Edition*) pues se trata de la versión gratuita y *opensource*.

A.1.1. Instalación Entorno Local

A.1.1.1. *docker*

Dado que Arch utiliza un sistema de gestión de paquetes diferente al más común *apt* usaremos *pacman*.

```
$ sudo pacman -Syu           #Actualizar índice de paquetes
$ sudo pacman -S docker      #Instalamos Docker
```

Es importante agregar nuestro usuario al grupo recién generado ***docker***, de esta manera evitaremos tener que escribir *sudo* cada vez que queramos ejecutar un comando docker.

```
sudo usermod -aG docker $USER
```

A.1.1.2. docker-compose

Para `docker-compose` no tenemos el mismo problema pues lo sacamos directamente del repositorio *GitHub* oficial de Docker

```
#Descargamos binario
$ sudo curl -L "https://github.com/docker/compose/releases/
↳ download/1.25.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
#Damos permisos de ejecución
$ sudo chmod +x /usr/local/bin/docker-compose
```

A.1.2. Instalación Entorno Servidores/Cloud

Podremos instalar manualmente en cada uno de los nodos la versión última de Docker como vimos en la Subsubsección A.1.1.1 o utilizando un *script* creado por la comunidad de *GitHub* como vemos en la Figura C.5.

Para la instalación de `docker-compose` deberemos realizar el mismo procedimiento descrito en la Subsubsección A.1.1.2.

A.1.3. Uso de Docker y docker-compose

Comando básicos que tendremos que conocer para este trabajo.

```
##### docker #####

#Construir imagen a partir de un Dockerfile
$ docker build [OPTIONS] PATH
#Agregar un TAG a una imagen existente
$ docker tag IMAGE_ID NEW_TAG
#Descargar una imagen IMAGE:TAG del repositorio de Docker Hub
$ docker pull IMAGE:TAG
#Subir una imagen a un repositorio [reponame]
$ docker push reponame/imagenname:tag
#Ejecutar un comando COMMAND en el contenedor CONTAINER_NAME con configuraciones [OPTIONS]
$ docker exec [OPTIONS] CONTAINER_NAME COMMAND
#Conocer el consumo de recursos de los contenedores activos
$ docker stats
#Ejecuta un contenedor con las configuraciones [OPTIONS] y la imagen IMAGE
$ docker run [OPTIONS] IMAGE

##### docker-compose #####

# Estando en el mismo directorio que fichero docker-compose.yml lanzamos el servicio (detached)
$ docker-compose up -d
# Paramos y eliminamos los contenedores y redes asociadas al servicio del docker-compse.yml
$ docker-compose down
```

Mapeo de puertos en Docker Gracias a la opción `docker run -p EXT:INT` podremos mapear el puerto interno (INT deseado con un puerto externo (EXT) a nuestra máquina o también mediante la configuración `port: [EXT:INT]` en los ficheros `docker-compose.yml`. De esta manera podremos conseguir ejecutar, por ejemplo, dos servicios que dependan de `apache` (por defecto utiliza el puerto 80) sin tener que cambiar la configuración interna de la imagen. De lo contrario, los servicios no se podrían lanzar en la misma maquina por un conflicto de puertos (no puede haber dos aplicaciones diferentes sirviendo por el mismo puerto).

Mapeo de volúmenes en Docker Existen dos maneras diferentes de compartir datos entre la máquina local y el contenedor:

Mapeo directo Lo podemos asociar a compartir una carpeta igual que hacemos con las máquinas virtuales. Podemos hacerlo mediante el siguiente comando:

```
docker run -v PATH_LOCAL:PATH_CONTAINER,  
o en los ficheros docker-compose.yml mediante la configuración volumes:  
PATH_LOCAL:PATH_CONTAINER
```

Mapeo por volumen Este método no mapea un directorio local dentro del contenedor, si no que crea un **volumen** Docker que no es más que una forma propia de guardar la información. Podemos hacer que una ubicación del contenedor que nos interese se almacene en un volumen Docker para que, en caso de caída o eliminación del contenedor, cuando lo volvamos a desplegar exista **persistencia en los datos** de esa ubicación deseada. Es frecuente utilizarlo cuando desplegamos servicios tipo base de datos.

Esta opción la tenemos disponible mediante el uso del comando `docker run -v VOLUMEN_NOMBRE:PATH_CONTAINER` o en los ficheros `docker-compose.yml` mediante la configuración `volumes: VOLUMEN_NOMBRE:PATH_CONTAINER`

*Nota: En Linux podemos encontrar los volúmenes localizados en el directorio: `/var/lib/docker/volumes/` aunque tendremos que tener acceso **root** (super usuario).*

A.2. Instalación de TSRC

Esta herramienta se encuentra disponible en los repositorios de *Python*, por tanto podremos instalarlo de manera sencilla teniendo instalados los paquetes de `python3` y `python3-pip`.

```
$ pip3 install tsrc --user  
#Agregaremos ~/.local/bin a nuestro PATH  
export PATH=~/.local/bin:$PATH
```

A.3. Uso y configuración de TSRC

Para empezar a usar `tsrc` tendremos que configurarlo. Será tan sencillo como asignar la dirección de los diferentes repositorios de *GitLab* que clonaremos al principio en el fichero `manifest.yml` que se describe a continuación:

```
repos:
- src: ResultNode
  url: git@gitlab.com:marktfg/ResultNode.git
  branch: master
# (...)
```

Anexo B

Configuración de Herramientas

B.1. Configuración *.bashrc*

A continuación veremos la configuración que realizamos a nuestro entorno local y al del servidor para poder llamar a los diferentes *scripts* mediante *alias* y también a diferentes comandos.

B.1.0.1. Entorno Local

Aquí se muestra una posible configuración del entorno para facilitar el trabajo en local, pero puede ser modificada a libre elección.

```
export MARKTFG="/home/$USER/Documents/GIT/MarkTFG"
alias tfg-rebuild="$MARKTFG/DevOps/Scripts/Local/rebuildDeploy.sh"
alias tfg-down="$MARKTFG/DevOps/Scripts/Local/downDeploy.sh"
alias tfg-test="$MARKTFG/DevOps/Scripts/Local/GenerateVotes/requests.sh"
alias tfg-cleanVotes="$MARKTFG/DevOps/Scripts/Local/cleanVotes.sh"
alias tfg-pushImages="$MARKTFG/DevOps/Scripts/Local/push-images.sh"
source $MARKTFG/DevOps/Scripts/Local/script-completion-tfg.bash
```

B.1.0.2. Entorno en Servidores

Al igual que antes, en la Figura B.1 se muestra una sugerencia de configuración. Recordemos que, si se usara, solo es necesaria en nuestro nodo administrador (nodo manager).

```

export MARKTFG="/home/$USER/markTFG"
export DIR_HOSTS="/home/$USER/markTFG/DevOps/Scripts/Server/hosts"

alias tfg-deploy-web="docker network create -d overlay tfg; docker stack deploy -c
↳ $MARKTFG/DevOps/ServerConf/docker-compose.yml --with-registry-auth tfg"
alias tfg-rm-web="docker stack rm tfg"

alias tfg-deploy-monitoring="docker stack deploy -c
↳ $MARKTFG/DevOps/ServerConf/Monitoring/docker-compose.yml monitoring"
alias tfg-rm-monitoring="docker stack rm monitoring"

alias tfg-deploy-visualizer="docker stack deploy -c
↳ $MARKTFG/DevOps/ServerConf/Visualizer/docker-compose.yml visualizer"
alias tfg-rm-visualizer="docker stack rm visualizer"

alias tfg-deploy-jenkins="docker stack deploy -c
↳ $MARKTFG/DevOps/ServerConf/Jenkins/docker-compose.yml jenkins"
alias tfg-rm-jenkins="docker stack rm jenkins"

alias tfg-prune="docker system prune -f; parallel-ssh -t 0 -i -h $DIR_HOSTS/all '-x -i
↳ ~/.ssh/id_generic' 'docker system prune -f'"
alias tfg-node-update="parallel-ssh -t 0 -h $DIR_HOSTS/all -l root -x '-i ~/.ssh/id_generic'
↳ 'apt update'; parallel-ssh -t 0 -i -h $DIR_HOSTS/all -l root -x '-i ~/.ssh/id_generic'
↳ 'apt list --upgradable'"
alias tfg-node-fullupgrade="parallel-ssh -t 0 -h $DIR_HOSTS/all -l root '-x -i
↳ ~/.ssh/id_generic' 'DEBIAN_FRONTEND=noninteractive apt full-upgrade -y'"
alias tfg-node-reboot="parallel-ssh -h $DIR_HOSTS/all -l root -x '-i ~/.ssh/id_generic'
↳ 'reboot'"
alias tfg-node-images="parallel-ssh -t 0 -i -h $DIR_HOSTS/all '-x -i ~/.ssh/id_generic'
↳ 'docker image ls'; docker image ls"
alias tfg-update-image="$MARKTFG/DevOps/Scripts/Server/update-images.sh"
source $MARKTFG/DevOps/Scripts/Server/script-completion-tfg.bash

alias jenkins-vote="tfg-update-image votepython"
alias jenkins-result="tfg-update-image resultnode"

```

Figura B.1: Configuración `.bashrc` para el servidor

B.2. Configuración SSH

En esta sección mostraremos la configuración creada para poder acceder de manera rápida y segura a todos nuestros nodos que componen el *cluster*, así como la configuración realizada dentro del servidor.

B.2.0.1. Entorno Local

En nuestra máquina local configuraremos un “alias” a los servidores para conectarnos de manera rápida con el usuario no `root` creado en los servidores, que es: `tfgdenis`. Lo configuramos a través del fichero `~/.ssh/config` como vemos en la Figura B.2

También hemos tenido que crear las claves para poder copiarlas a nuestro servidor, para ello hemos ejecutado el comando `ssh-keygen` y hemos copiado la clave al servidor mediante el comando `ssh-copy-id` para agregar nuestra clave a la lista de claves autorizadas del servidor. Debemos hacer la copia en todos nuestros nodos, para poder tener acceso a ellos.


```

#Corresponde al nodo principal - administrador/manager
host tfgmaster
    HostName 167.71.141.65
    User tfgdenis
#Corresponde al nodo2 - worker
host tfg2
    HostName 167.71.139.249
    User tfgdenis
#Corresponde al nodo3 - worker
host tfg3
    HostName 167.172.54.37
    User tfgdenis

```

Figura B.2: Configuración del archivo SSH

B.2.0.2. Entorno en Servidores

Al igual que en el entorno local, debemos realizar el mismo proceso para poder conectarnos a los demás nodos de manera sencilla, agregando el fichero `~/.ssh/config`

```

Host gitlab.com
    Preferredauthentications publickey
    IdentityFile ~/.ssh/id_git
host node2
    HostName 167.71.139.249
    User tfgdenis
    IdentityFile ~/.ssh/id_generic
host node3
    HostName 167.172.54.37
    User tfgdenis
    IdentityFile ~/.ssh/id_generic

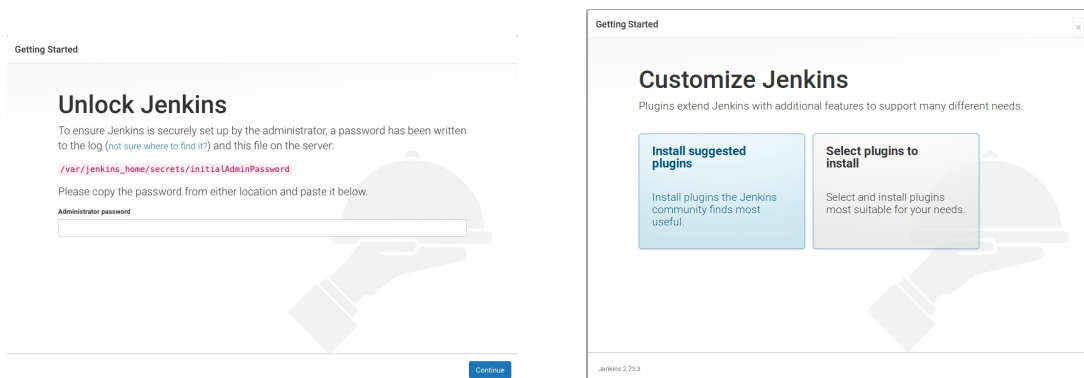
```

También deberemos crear una clave mediante `ssh-keygen` y copiarla con `ssh-copy-id`. Además, una vez realizado el proceso de copia (el cual nos pide la contraseña de usuario al nodo que nos estamos conectando) modificaremos la configuración del servicio para que no sea posible conectarse mediante usuario y contraseña y que solo sea posible mediante clave SSH; para ello modificaremos el archivo `/etc/ssh/sshd_config` con los siguientes parámetros:

```

PermitRootLogin no
PasswordAuthentication no
#Configuración mantener sesión activa (envió paquete vacío)
TCPKeepAlive yes
ClientAliveInterval 60
ClientAliveCountMax 10000

```



(a) Contraseña inicial por defecto

(b) Instalación de paquetes sugeridos

Figura B.3: Configuración inicial

B.3. Configuración Jenkins

Explicaremos brevemente algunas de las configuraciones básicas para poder realizar las tareas dentro del servicio `jenkins`, así como configuraciones en los demás servicios que dependan.

B.3.1. Configuración inicial

Tras lanzar el contenedor, *Jenkins* se iniciará y nos pedirá que introduzcamos la contraseña inicial establecida automáticamente por el servicio. Para esto tendremos que lanzar en nuestro nodo el comando `docker exec -it CONTENEDOR_JENKINS cat /var/jenkins_home/secret/initialAdminPassword`

Una vez realizado el cambio de contraseña instalaremos los paquetes por defecto. Ver la Figura B.3

Tras la instalación de los paquetes por defecto instalaremos algunos que son imprescindibles para nuestras tareas, estos son:

- **Docker** Para poder utilizar el *socket* de Docker
- **GitLab** Para poder utilizar la *API* y los *Web-hooks*
- **Build With Parameters** Parametrizar la construcción en las tareas.
- **Exclusive Execution** Asegura la ejecución de tareas una a una.
- **Date Parameter Plugin** Crea una variable con la fecha.

Por último agregaremos una *nube* que nos servirá para conectar con el `socket` de Docker, como vemos en la figura Figura B.4

Cloud

Docker

Name:

Docker Host URI:

Server credentials: - none - Add

Advanced...

Test Connection

Figura B.4: Agregamos socket Docker del nodo en la que estamos ejecutando Jenkins

Jenkins Credentials Provider: Jenkins

Add Credentials

Domain: Global credentials (unrestricted)

Kind: SSH Username with private key

Scope: Global (Jenkins, nodes, items, all child items, etc)

ID:

Description:

Username:

Private Key: ☒ Enter directly

Key:

Passphrase:

Figura B.5: Configuración credenciales Jenkins SSH con GitLab

B.3.2. Configuración Gitlab

Esta sección explica todo lo necesario para trabajar de manera segura con *GitLab*

B.3.2.1. Configuración SSH

Debemos configurar la clave SSH como haríamos para cualquier otra máquina; tendremos que conectarnos manualmente al contenedor de `jenkins` mediante el comando `docker exec -it CONTENEDOR_JENKINS bash` y realizar los comandos explicados en la Sección B.2.

Una vez configurada la clave SSH, ahora tendremos que agregar a las credenciales de *Jenkins* una credencial de tipo **SSH Username with private key** para poder realizar las clonaciones de repositorios dentro de las tareas, como se ve en la Figura B.5. Además deberemos agregar la clave pública configurada a GitLab. Lo explicamos en la Sección C.1.

Gitlab

Enable authentication for '/project' end-point ☒

GitLab connections

Connection name

A name for the connection

Gitlab host URL

The complete URL to the Gitlab server (e.g. http://gitlab.mydomain.com)

Credentials

API Token for accessing Gitlab

Figura B.6: Configuración GitLab API

Jenkins Credentials Provider: Jenkins

Domain

Kind

Scope

Username

Password

ID

Description

Figura B.7: Configuración credenciales Docker Hub con usuario y contraseña

B.3.2.2. Configuración GitLab API

Una vez establecida la clave SSH, ahora tendremos que configurar la conexión a *GitLab* mediante la *API*, para ello tendremos que configurar el paquete extra que instalamos anteriormente de *GitLab* y agregar además las credenciales SSH que configuramos en el paso anterior. Véase la Figura B.6.

B.3.3. Configuración Docker Hub

La configuración será mucho mas simple que el caso de *GitLab* pues únicamente deberemos agregar a las credenciales de *Jenkins* usuario y contraseña como se muestra en la Figura B.7.

Anexo C

Configuración de Servicios

C.1. Configuración GitLab

En primer lugar necesitamos crear un **grupo**, para tener los diferentes proyectos que se estén llevando a cabo, bien organizados y separados. En nuestro caso este grupo lo hemos llamado **MarkTFG** y en él hemos creado todos los repositorios que necesitamos para la realización de este trabajo. Ver la Figura C.1.

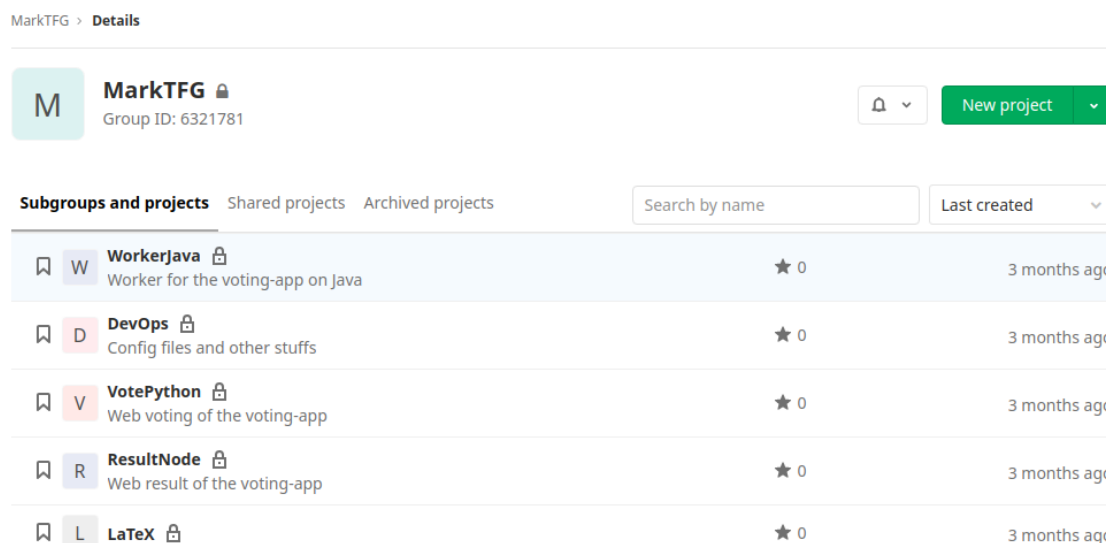


Figura C.1: Repositorios creados para el trabajo

C.1.1. Configuración SSH

Como hemos visto en la Apéndice B para utilizar *Jenkins* de una manera segura tendremos que tener a disposición claves SSH. Una vez agregadas a *Jenkins*, configuraremos *GitLab* simplemente agregando la clave pública en la configuración de

éste (Figura C.2).

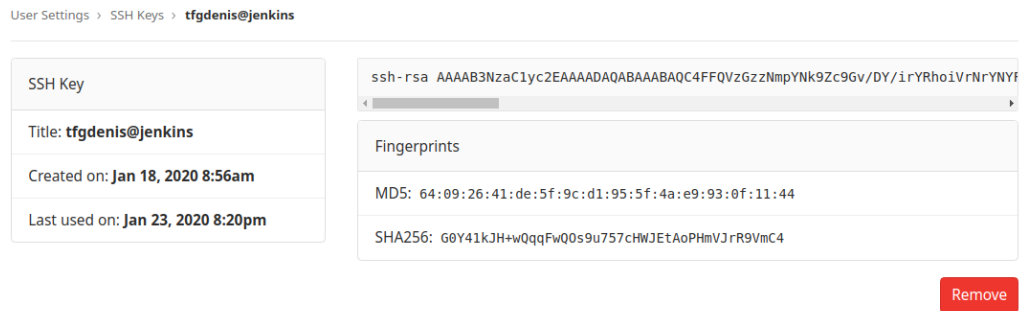


Figura C.2: Incorporación de la clave pública SSH de Jenkins

C.1.2. Configurar WebHooks

Es una parte vital para la Integración Continua pues esta configuración será la encargada de enviar una alerta a nuestro servicio de *Jenkins* desde *GitLab* nada más se suban cambios al repositorio en cuestión. Para realizar esta configuración deberemos dirigirnos al repositorio que queremos dotar de esta funcionalidad y dirigirnos a **Settings->Integrations** una vez allí rellenaremos los campos de **URL** y **Secret Token** con lo que *Jenkins* nos diga en la configuración de la tarea. Deberemos obtener una entrada como la que vemos en la Figura C.3.

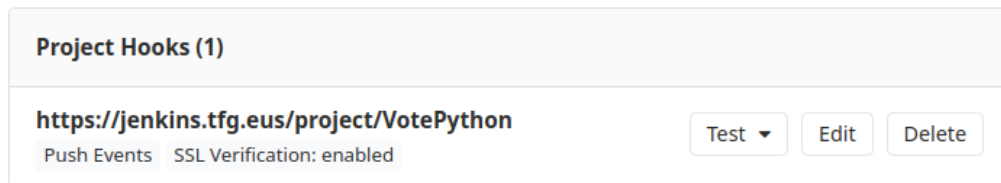


Figura C.3: Agregado WebHook para servicio votos de la aplicación

C.2. Configuración DockerHub

Es muy importante tener en cuenta que se han usado repositorios públicos, pues son gratuitos, pero cualquiera podría ver nuestros códigos. En caso de requerir un repositorio privado será necesario asumir un coste. Además deberemos tener en cuenta que todas nuestras imágenes tendrán como prefijo el nombre de nuestro Docker ID, pues todos los repositorios comenzarán con este nombre. En la Figura C.4 podemos ver la estructura que seguimos para las diferentes imágenes que usaremos en el trabajo.

tfgdennis	Search by repository name...	Create Repository
tfgdennis / dev_resultnode Updated 8 days ago	☆ 0	↓ 196 PUBLIC
tfgdennis / dev_votepython Updated 8 days ago	☆ 0	↓ 226 PUBLIC
tfgdennis / dev_worker Updated 16 days ago	☆ 0	↓ 119 PUBLIC
tfgdennis / prod_worker	☆ 0	↓ 0 PUBLIC
tfgdennis / prod_resultnode	☆ 0	↓ 0 PUBLIC
tfgdennis / prod_votepython	☆ 0	↓ 0 PUBLIC

Figura C.4: Repositorios de todas las imágenes disponibles para este trabajo

C.3. Configuración DigitalOcean

Además de poder elegir el sistema operativo con el que queremos dotar a cada uno de nuestros nodos (Ubuntu 18.04 en nuestro caso) y la región en la que se aloja (Londres), nos interesa destacar las configuraciones iniciales utilizadas pues nos ahorrarán mucho tiempo después. La más importante será la instalación de Docker y esto lo podemos hacer a través de un *script* de libre uso. También será importante el nombre que usemos para nuestros nodos. (ver Figura C.5).

C.4. Configuración DinaHosting

En el panel de configuración editaremos el **Servidor DNS** para que en lugar de utilizar el predeterminado, utilizamos el de *DigitalOcean*, como vemos en la Figura C.6.

Select additional options ?

☐ Private networking
 ☐ IPv6
 ☒ User data
 ☐ Monitoring

```
#Install Docker
runCmd:
- curl -fsSL https://get.docker.com/ | sh
```

Authentication ?

☒ **SSH keys**
A more secure authentication method

☐ **One-time password**
Emails a one-time root password to you (less secure)

☒ Select all
 ☒ mark@xiaomi

[New SSH Key](#)

How many Droplets?

Deploy multiple Droplets with the same [configuration](#).

—
3 Droplets
+

Node-01

Node-02

Node-03

Choose a hostname

Give your Droplets an identifying name you will remember them by. Your Droplet name can only contain alphanumeric characters, dashes, and periods.

Figura C.5: Configuración inicial del Cluster

- [Inicio](#)
- [Contactos](#)
- [Servidores DNS](#)
- [Zonas DNS](#)
- [Redirecciones](#)
- [Cuenta de email](#)
- [Parking](#)
- [Panel de acceso](#)

DNS por defecto

☐ Quiero utilizar los DNS de dinahosting

DNS propios

DNS 1

DNS 2

DNS 3

DNS 4

DNS 5

DNS 6

[Guardar](#)

Figura C.6: Establecemos Servidor DNS de DigitalOcean

Agradecimientos

Quisiera agradecer a Eladio, por transmitirme su vocación a lo largo de la carrera desde que tuve el placer de conocerle.

Gracias a mis padres y mi hermana por darme todo el apoyo necesario para seguir adelante con mi formación y no desistir conmigo.

Por último, gracias a Cristina, por toda la ayuda y la dedicación que me ha dado a lo largo de este proyecto.

Referencias

- admin. (2019, noviembre). *What is Prometheus?* Descargado 2020-01-28, de <http://www.techplayon.com/what-is-prometheus/>
- Atlassian. (s.f.). *Continuous integration vs. continuous delivery vs. continuous deployment.* Descargado 2020-01-26, de <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>
- Database Scaling : Horizontal and Vertical Scaling.* (s.f.). Descargado 2020-01-26, de <https://hackernoon.com/database-scaling-horizontal-and-vertical-scaling-85edd2fd9944>
- DINAHOSTING / Dominios.es.* (s.f.). Descargado 2020-02-06, de <https://www.dominios.es/dominios/es/agentes-registradores/dinahosting-dockersamples/example-voting-app>
- dockersamples/example-voting-app.* (2020, febrero). Docker Samples. Descargado 2020-02-06, de <https://github.com/dockersamples/example-voting-app> (original-date: 2015-11-15T19:59:56Z)
- Escalabilidad.* (2020, enero). Descargado 2020-01-26, de <https://es.wikipedia.org/w/index.php?title=Escalabilidad&oldid=122808308> (Page Version ID: 122808308)
- Getting started with swarm mode.* (2020, enero). Descargado 2020-01-27, de <https://docs.docker.com/engine/swarm/swarm-tutorial/>
- GitLab.* (2020, enero). Descargado 2020-01-29, de <https://en.wikipedia.org/w/index.php?title=GitLab&oldid=937642531> (Page Version ID: 937642531)
- How nodes work.* (2020, enero). Descargado 2020-01-29, de <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>
- Integración Continua de Software: Jenkins, mayordomo a nuestro servicio.* (2018, abril). Descargado 2020-01-28, de <https://pandorafms.com/blog/es/integracion-continua/>
- An introduction to Git: what it is, and how to use it.* (2018, agosto). Descargado 2020-01-28, de <https://www.freecodecamp.org/news/what-is-git-and-how-to-use-it-c341b049ae61/>
- LaTeX.* (2020, enero). Descargado 2020-01-28, de <https://en.wikipedia.org/w/index.php?title=LaTeX&oldid=934628926> (Page Version ID: 934628926)
- Nginx.* (2020, enero). Descargado 2020-01-28, de <https://en.wikipedia.org/w/index.php?title=Nginx&oldid=935411914> (Page Version ID: 935411914)
- Node.js.* (2019, diciembre). Descargado 2020-01-28, de <https://es.wikipedia.org/w/index.php?title=Node.js&oldid=122223864> (Page Version ID:

122223864)

Python (programming language) - Wikipedia. (s.f.). Descargado 2020-01-28, de [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))

Rodríguez, L. (s.f.). *Containers - Contenedores Informáticos*. Descargado 2020-01-26, de <https://www.teamnet.com.mx/blog/containers-contenedores-informáticos>

Security, A. (s.f.). *Container, Serverless & Cloud Native Application Security*. Descargado 2020-01-26, de <https://blog.aquasec.com/>

TankerHQ/tsrc. (2020, enero). Tanker. Descargado 2020-01-28, de <https://github.com/TankerHQ/tsrc> (original-date: 2017-07-21T12:29:59Z)

Una Breve Historia de los Contenedores: desde los años 70 a Docker 2016. (2017, octubre). Descargado 2020-01-26, de <https://hoplasoftware.com/una-breve-historia-de-los-contenedores-desde-los-anos-70-a-docker-2016/>

Understanding Bash: Elements of Programming | Linux Journal. (s.f.). Descargado 2020-01-28, de <https://www.linuxjournal.com/content/understanding-bash-elements-programming>

Virtualización. (2020, enero). Descargado 2020-01-26, de <https://es.wikipedia.org/w/index.php?title=Virtualizaci%C3%B3n&oldid=122628781> (Page Version ID: 122628781)

What is a Container? (s.f.). Descargado 2020-01-26, de <https://www.docker.com/resources/what-container>

What is DigitalOcean and why should you host apps on it? (s.f.). Descargado 2020-01-29, de <https://www.cloudways.com/blog/what-is-digital-ocean/>

What is Docker Hub? - Definition from WhatIs.com. (s.f.). Descargado 2020-01-29, de <https://searchitoperations.techtarget.com/definition/Docker-Hub>

What is Docker Swarm? (s.f.). Descargado 2020-01-28, de <https://www.sumologic.com/glossary/docker-swarm/>

What is GitLab? (s.f.). Descargado 2020-01-29, de <https://about.gitlab.com/what-is-gitlab/>

What is Grafana? Why Use It? Everything You Should Know About It. (2019, enero). Descargado 2020-01-28, de <https://www.8bitmen.com/what-is-grafana-why-use-it-everything-you-should-know-about-it/>

What is High Availability? - Definition from WhatIs.com. (s.f.). Descargado 2020-01-26, de <https://searchdatacenter.techtarget.com/definition/high-availability>

What is Java programming language? - HowToDoInJava. (s.f.). Descargado 2020-01-28, de <https://howtodoinjava.com/java/basics/what-is-java-programming-language/>

What is Linux? (s.f.). Descargado 2020-01-28, de <https://www.linux.com/what-is-linux/>

Working With Docker Hub. (s.f.). Descargado 2020-01-29, de <https://www.aquasec.com/wiki/display/containers/Working+With+Docker+Hub>



UNIVERSIDAD
DE MÁLAGA

| **uma.es**

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga